

IBM RT PC

VS Pascal User's Guide

Programming Family



Personal
Computer
Software

SH23-0127



IBM RT PC

VS Pascal User's Guide

Programming Family



Personal
Computer
Software

First Edition (March 1987)

The information in this manual applies to Version 1.0 of IBM RT PC VS Pascal and Version 1.0 of IBM RT PC VS FORTRAN for use with Release 2.1 of the AIX Operating System, and it applies to all subsequent releases and modifications until otherwise indicated in new editions or Technical Newsletters.

Changes are made periodically to the information herein; these changes will be incorporated in new editions of this publication.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM program product in this publication is not intended to state or imply that only IBM's program product may be used. Any functionally equivalent program may be used instead.

International Business Machines Corporation provides this manual "as is" without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this manual at any time.

Requests for copies of this product and for technical information about the system should be made to your authorized IBM RT PC dealer.

A reader's comment form is provided at the back of this publication. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

© IBM Corporation 1987

™RT PC is a trademark of IBM Corporation.

™AIX is a trademark of IBM Corporation.

Preface

This manual is a user's guide for developing and executing Pascal programs on the IBM RT PC¹ using the AIX² Operating System.

This manual is intended for users who have a good understanding of the Pascal programming language. A detailed presentation is provided in the *RT PC VS Pascal Reference Manual*. Users should also be familiar with the AIX Operating System.

Contents

Chapter 1 — "Introduction" includes the highlights of the RT PC VS Pascal language and the conventions used in this guide.

Chapter 2 — "The Compiler" provides the information necessary to compile Pascal programs and describes each of the available command-line options and compiler directives.

Chapter 3 — "Using Input and Output Facilities" describes the procedures for opening files for input and output on the AIX Operating System.

Chapter 4 — "Data Representations" describes how IBM RT PC VS Pascal represents data storage.

Chapter 5 — "Mixing Languages" describes the procedures required when mixing program elements written in various RT PC languages such as Pascal, FORTRAN and C. It also illustrates the mechanism for passing parameters to subroutines and functions.

¹ RT PC is a trademark of IBM Corporation.

² AIX is a trademark of IBM Corporation.

Chapter 6 — "The Disassembler" describes how to translate binary code modules into assembly language equivalents.

Appendix A — "Messages" lists the compile-time and run-time error messages.

Appendix B — "ASCII Character Set" lists the decimal, octal, and hexadecimal values for the American National Standard ASCII characters.

Appendix C — "Migrating Programs" describes the information necessary to compile IBM mode and ANSI mode source programs.

Related Publications

You may want to consult the following RT PC publications for additional information:

- *VS Pascal Reference Manual*, SH23-0128, describes the Pascal data types, expressions, program structures, and standard procedures and functions as implemented on the RT PC.
- *VS FORTRAN User's Guide*, SH23-0129, describes the procedures for compiling and running FORTRAN programs using RT PC FORTRAN, Version 2, with the AIX Operating System.
- *VS FORTRAN Reference Manual*, SH23-0130, describes the high-level FORTRAN 77 language as implemented on the RT PC.
- *VS Language/Operating System Interface Library*, SH23-0131, describes the system routines which can be called from Pascal and FORTRAN programs.
- *Concepts*, GC23-0784, provides an overview of the RT PC hardware, the AIX Operating System, including system management, communications, applications, and supporting publications.

- *Installing and Customizing the AIX Operating System*, SV21-8001, provides step-by-step instructions for installing and customizing the AIX Operating System, including how to add or delete devices from the system and how to define device characteristics. This book also explains how to create, delete, or change AIX and non-AIX minidisks.
- *Messages Reference*, SV21-8002, lists messages displayed by the RT PC and explains how to respond to the messages.
- *Usability Services Guide* and *Usability Service Reference*, SV21-8003, shows how to create and print text files, work with directories, start application programs, and do other basic tasks with Usability Services.
- *Using and Managing the AIX Operating System*, SV21-8004, contains information on using AIX Operating System commands, working with the file system, developing shell procedures, and performing such system management tasks as creating and mounting file systems, backing up the system, and repairing file system damage.
- *AIX Operating System Commands Reference*, SV21-8005, lists and describes the AIX Operating System commands.
- *C Language Guide and Reference*, SV21-8008, provides information for writing, compiling, and running C language programs.
- *AIX Operating System Technical Reference*, SV21-8009, describes the system calls and subroutines a programmer uses to write programs. This book also provides information about the AIX file system, special files, miscellaneous files, as well as information on writing device drivers.
- *AIX Operating System Programming Tools and Interfaces*, SV21-8010, describes the programming environment of the AIX Operating System and includes information about using the operating system tools to develop, compile, and debug programs.
- *AIX Operating System DOS Services Reference*, SV21-8012, provides step-by-step information for using the AIX Operating System Shell. In addition, this book describes the DOS system services.
- *User Setup Guide*, SV21-8020, provides instructions for setting up and connecting devices to system units. It also gives procedures for installing the AIX Operating System and for testing the setup.

- *Guide to Operations*, SV21-8021, describes system units, the displays, keyboard, and other devices that can be attached. This guide also includes procedures for operating the hardware and moving system units.
- *Problem Determination Guide*, SV21-8022, provides instructions for running diagnostic routines to locate and identify hardware problems. It also includes problem determination procedures for software.

You may want to consult the following IBM publications for additional information:

- *Pascal/VS Language Reference Manual*, SH20-6168, describes the implementation of the Pascal/VS compiler.
- *Pascal/VS Programmer's Guide*, SH20-6162, describes how to compile and execute Pascal/VS programs.

Contents

Chapter 1. Introduction	1-1
Compiler Modes	1-2
Pascal Programs Under AIX	1-2
Compilation Process	1-4
Methods of Presentation	1-6
 Chapter 2. The Compiler	 2-1
Invoking the Compiler	2-1
Command-Line Options	2-3
Optimization of Programs	2-7
Compiler Directives	2-9
 Chapter 3. Using Input and Output Facilities	 3-1
Opening Files for Input and Output	3-1
Environment-Determined Files	3-2
Using Environment Variables on the Command Line	3-2
Using Environment Variables in Shell Scripts	3-6
Program-Determined Files	3-7
Open Options	3-7
Terminal Input and Output	3-10
 Chapter 4. Data Representations	 4-1
Storage Allocation	4-1
Representation of Integers	4-3
Representation of Shortreals	4-3
Representation of Reals	4-4
Representation of Extreme Values	4-5
Hexadecimal Representation of Selected Numbers	4-6
Representation of Sets	4-6
Representation of Arrays	4-7
Representation of Pointers	4-8
Representation of Strings	4-8
Packing Methods	4-8
 Chapter 5. Mixing Languages	 5-1

Correspondence of Data Types	5-1
Storage of Matrices	5-3
Input/Output Primitives	5-4
Subroutine Linkage Convention	5-5
Load Module Format	5-5
Register Usage	5-5
Stack Frame	5-8
Parameter Passing	5-11
Function Values	5-12
Parameter Addressing	5-12
Traceback	5-12
Entry and Exit Code	5-12
Calling a Routine	5-13
Sample Programs	5-13
Pascal Calling FORTRAN and C	5-14
FORTRAN Calling Pascal and C	5-18
C Calling FORTRAN and Pascal	5-22
 Chapter 6. The Disassembler	6-1
Preparation	6-1
Automatic Option Memory File	6-2
Using the Disassembler	6-2
From the Command Line — with Options	6-2
From the Command Line — without Options	6-9
From the Menu System	6-12
From a Command File	6-18
 Appendix A. Messages	A-1
Compile-Time Lexical Messages	A-1
Compile-Time Syntactic Messages	A-2
Compile-Time Semantic Messages	A-3
Compiler Limitation Messages	A-6
Input/Output Messages	A-7
 Appendix B. ASCII Character Set	B-1
 Appendix C. Migrating Programs	C-1
 Index	X-1

Chapter 1. Introduction

IBM RT PC VS Pascal provides a high performance optimizing compiler that produces object code for execution on the RT PC under the AIX Operating System. RT PC VS Pascal accepts Pascal source code as defined by IBM Pascal VS and the ANSI-83 standard. It offers significant performance improvement as well as many new compiler functions.

In addition to excellent performance, IBM RT PC VS Pascal offers these enhanced functions:

- Source compatibility with IBM Pascal/VS Release 2.2¹
- Source compatibility with ANSI X3.97-1983 standard
- Automated installation
- Fast compile performance
- Highly optimized code
- High usability
- Operating system interface library
- Common development/debugging environment
- Easy interlanguage linkages with FORTRAN and C
- Detailed on-screen messages
- Disassembler for user-readable object code

¹ See Appendix C, "Migrating Programs" for exceptions.

Compiler Modes

RT PC VS Pascal offers two modes for compiling a program: IBM mode and ANSI mode. This feature can be used as an effective development tool since it allows programs to be written and tested on independent RT PC workstations and then moved to a mainframe that uses Pascal VS. You may mix modes in creating an executable program; however, each separate compilation unit may be only a single mode. You may work in the mode you need or with which you are most familiar.

IBM Mode

This mode is the default mode of the compiler. It allows you to compile code that uses the IBM Pascal/VS Release 2.2 definition of Pascal. All Pascal statements, data types, math libraries, and directives are supported.

ANSI Mode

This mode uses only the ANSI X3.97-1983 definition of Pascal. When this mode is used, only ANSI-83 standard syntax programs are compiled. ANSI mode may be selected by entering the appropriate command-line option when invoking the compiler. For a description of the command-line options, see Chapter 2, "The Compiler."

Pascal Programs Under AIX

As illustrated in Figure 1-1, the four main steps in developing an executable Pascal program on the AIX Operating System are:

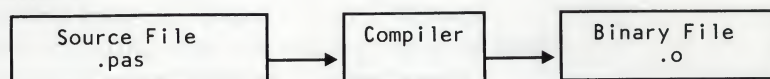
1. Create your program using a text editor and save it with a ".pas" extension.
2. Compile your source program to generate a binary file.

3. Link the output with the AIX system linker "cc" to create an executable file.
 4. Run the program.
-

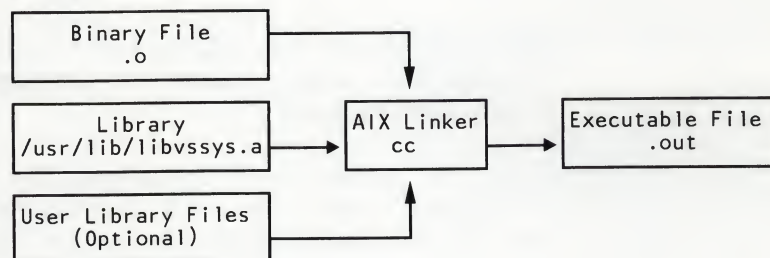
STEP 1



STEP 2



STEP 3



STEP 4

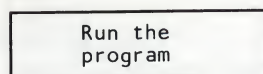


Figure 1-1. Preparing a Pascal Program under AIX

Compilation Process

As illustrated in Figure 1-2, the compiler follows these steps when invoked:

1. The source file is passed to "vspascal", which produces intermediate code with an ".i" extension.
2. The ".i" file is passed to "vspass2", the code generator.
3. Code is then passed to "vspass3".
 - a. If the "g+" command-line option was set when the code was passed to "vspascal", then "vspass3" creates both a ".dbg" file that may be used with the Disassembler (disasm) and a binary file (.o file). The ".o" file is passed to the AIX linker (cc) which creates an executable file. The executable file may be debugged by the Symbolic Debugger (sdb).

Note: If both the "d+" and the "g+" command-line options are set, regardless of their order on the command line, the "g+" option has the higher priority.

- b. If the "d+" command-line option was set when the code was passed to "vspascal", then "vspass3" creates both a ".dbg" file that may be used with the Disassembler (disasm) and a binary file (.o file). The ".o" file is passed to the AIX linker (cc) which creates an executable file. The executable file may not be debugged by the Symbolic Debugger (sdb).
 - c. If neither command-line option was set when the code was passed to "vspascal", then "vspass3" creates a binary file (.o file). The ".o" file is passed to the AIX linker (cc) which creates an executable file. The executable file may not be debugged by the Symbolic Debugger (sdb).

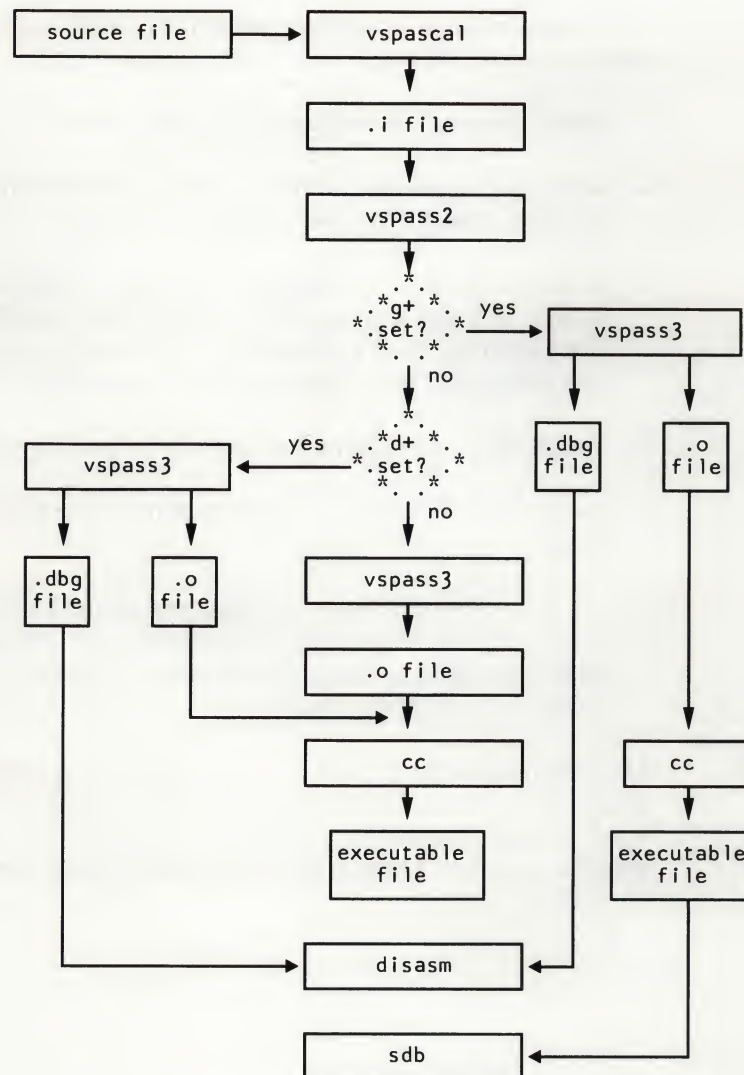


Figure 1-2. The Compilation Process

Methods of Presentation

The notations used to explain the RT PC VS Pascal language syntax in this manual are as follows:

- **bold** letters and words appear as they should in programs.
- *italicized* letters and words indicate a variable that should be substituted by data objects in actual program statements.
- a double period (..) meaning "through", indicates an ordered sequence where only the start and end elements are specified (for example, packed array [1..5] of boolean). The double period (..) should not be confused with the ellipsis used in mathematics and writing.
- brackets([]) indicate optional items and subscripts of an array.
- braces ({ }) enclose elements that can be repeated "zero to many times".

Note: Although braces are also one of the forms of comment delimiters in RT PC VS Pascal, this should not cause any difficulty. The one case where misinterpretation might occur is in the definition of comments, and this is explicitly shown.

Occasionally, it is necessary to use a specific character which results from pressing two keys at the same time. For example, the RT PC VS Pascal **eof** (end-of-file) character is "Ctrl-D". To enter this character in a file, both the Ctrl and the D keys must be pressed simultaneously.

Chapter 2. The Compiler

Pascal source code is compiled on the RT PC by executing the "vsp" compiler, which produces binary code. The binary code is then linked using the AIX Operating System linker, "cc".

This chapter describes how to execute the compiler and includes a description of each of the available command-line options and compiler directives.

Note: Additional information on the AIX Operating System linker may be found in the manual *AIX Operating System Commands Reference*.

Invoking the Compiler

The format for running the compiler from the command line is:

vsp *sourcefl* [*option*]...

sourcefl

is the name of a source file that has a ".pas" extension. If the extension is not specified, the compiler searches for a "*sourcefl*.pas" file. This is the only argument required for the compiler's operation.

option

is any of the command-line options listed in "Command-Line Options." They can appear in any order but must be separated by a space. If no options are specified, the compiler:

- sets margins at 1 and 72
- specifies 60 lines per page
- writes error messages to the standard output device
- generates calls to a compatible software library
- compiles in IBM mode
- specifies sequence fields at 73 and 80
- produces warning messages.

The following Pascal program and commands illustrate the procedures for invoking the RT PC VS Pascal compiler.

Example:

```
program sample;
begin
  writeln('Hello from Pascal!');
end.
```

To compile this program, enter `vsp sample`. The command invokes a shell script named "vsp" which runs the compiler and appends a `.pas` to the program name supplied.

The screen displays the message:

```
0 errors 0 warnings. 4 lines. File sample.pas
```

This program now runs whenever `sample` is entered. The output from this Pascal source program is:

```
Hello from Pascal!
```

Note: The program does not run if you enter `sample.pas`. An error message is produced stating that the program cannot be executed.

Command-Line Options

Command-line options indicate which features are enabled or disabled when the compiler is invoked. The options are described in alphabetical order. Figure 2-1 lists all the command-line options and their functions.

bm,n

MARGINS

sets the left and right margins of the source program. The compiler scans each line of the program starting at column *m* and ending at column *n*. Any data outside these margin limits is ignored. The maximum right margin allowed is 100. The specified margins must not overlap the sequence field as specified by command-line option *s*. If *n* is greater than 72, then this option must be followed by *s-* to disable the sequence field.

The default is *b1* and 72.

cn

LINES PER PAGE

specifies the number of lines to appear on each page of the output listing. The maximum number of lines to fit on a page depends on the type of printed form being used.

The default is *c60* or 60 lines to the page.

d+

DISASSEMBLER INFORMATION

instructs the compiler to put disassembler information in the ".dbg" file. This option is required if the module is to be disassembled using the RT PC Disassembler. This option also prepares the binary file for profiling. For more information on profiling, see the *AIX Operating System Commands Reference*.

efilename

ERROR FILE

instructs the compiler to place its error output in *filename*. If this option is not specified, the error output is directed to the standard output device.

f+	FLOATING-POINT HARDWARE instructs the compiler to generate in-line calls to floating-point hardware. If this option is used, the hardware feature is required at run time, but it is optional at compile time. The default instructs the compiler to generate calls to a compatible software library.
g+	DEBUGGER INFORMATION instructs the compiler to put debugger information in the executable file. This option is required if this module is to be debugged using the RT PC Symbolic Debugger (sdb).
l+	LIST TO STANDARD OUTPUT DEVICE instructs the compiler to generate a listing to the standard output device.
lfilename	LISTING FILE instructs the compiler to place its listing output in <i>filename</i> . If this option is not specified, no listing is generated.
man	ANSI MODE specifies ANSI mode which permits only the ANSI definition of Pascal. The use of any extension to the ANSI definition is flagged as a compile-time error.
mibm	IBM MODE specifies IBM mode, the default mode of the compiler. This mode permits only the Pascal/Vs Release 2.2 definition of Pascal. The use of any extension not in Pascal/Vs Release 2.2 is flagged as a compile-time error. This option is the default mode.
o1+	OPTIMIZATION LEVEL 1 instructs the compiler to use optimization level 1. Optimization is described in "Optimization of Programs."
o2+	OPTIMIZATION LEVEL 2 instructs the compiler to use optimization level 2. Optimization is described in "Optimization of Programs."

- o3+** **OPTIMIZATION LEVEL 3**
instructs the compiler to use optimization level 3. Optimization is described in "Optimization of Programs."
- o4+** **OPTIMIZATION LEVEL 4**
instructs the compiler to use optimization level 4. Optimization is described in "Optimization of Programs."
- sm,n** **SEQUENCE FIELDS**
specifies which columns of the program being compiled are reserved for a sequence field. The starting column of the sequence field is *m* and the last column of the sequence field is *n*.
- The minimum length of the sequence field is 2, as shown in the examples: s73,74 or s82,83. The sequence field must not overlap the margins of the compiler. See command-line option **b** on margins.
- The default is s73 and 80.
- The compiler does not process sequence fields; they are only used to identify lines in the source listing. If the sequence field is blank, the compiler inserts a line number in the corresponding area in the source listing.
- s-** **NO SEQUENCE FIELDS**
indicates that there is to be no sequence field.
- v+** **COMPILER PROGRESS INFORMATION**
instructs the compiler to generate information on the progress of the compile.
- w-** **NO WARNING MESSAGES**
instructs the compiler to suppress all warning messages produced by the compiler.

In any instance where a command-line option conflicts with a compiler directive, the compiler directive prevails. For example, if a **d+** option is specified on the command line and the **\$D-** compiler directive is also used, no disassembler information is generated. See "Compiler Directives" for more information on compiler directives.

Option	Function
<i>bm,n</i>	Margins
<i>cn</i>	Lines per page
d+	Disassembler information
<i>efilename</i>	Error file
f+	Floating-point hardware
g+	Debugger information
l+	List to standard output device
<i>lfilename</i>	Listing file
man	ANSI mode
mibm	IBM mode
o1+	Optimization level 1
o2+	Optimization level 2
o3+	Optimization level 3
o4+	Optimization level 4
<i>sm,n</i>	Sequence fields
s-	No sequence fields
v+	Compiler progress information
w-	No warning messages

Figure 2-1. Compiler Command-Line Options (Part 1 of 2).

Optimization of Programs

"Optimization" refers to the process of improving the execution performance of a given program. It is done at the cost of compile time but results in reduced execution time. The RT PC VS Pascal compiler performs two separate optimization passes — machine-dependent optimizations and machine-independent optimizations — which are controlled by selecting compiler command-line options. The command-line options for optimization are:

o1+ MACHINE-DEPENDENT OPTIMIZATION

instructs the compiler to perform a machine-dependent optimizing pass, which takes place after the code-generation phase of the compilation. This pass examines object code at the basic block level and includes:

- eliminating unnecessary branches
- eliminating redundant loads and stores
- exploiting machine idioms
- replacing branches with branch-with-execute instructions
- strength reduction.

o2+ MACHINE-INDEPENDENT OPTIMIZATION

instructs the compiler to perform a machine-independent optimizing pass that includes:

- constant folding
- straightening
- eliminating unreachable code
- copy propagation
- eliminating dead code.

o3+ MACHINE-INDEPENDENT OPTIMIZATION

instructs the compiler to perform a machine-independent optimizing pass that includes:

- eliminating common subexpressions
- subscript optimization
- eliminating induction variables
- loop invariant code motion.

o4+ MACHINE-DEPENDENT AND MACHINE-INDEPENDENT OPTIMIZATION

instructs the compiler to perform machine-dependent and machine-independent optimization.

Optimization Considerations

The choice of algorithm for a given task can have a much greater impact on execution speed than any compiler optimization. It is generally true that most of program execution time is spent on less than 10% of the code. Changes to the algorithm in the critical 10% frequently have dramatic results.

The optimizing feature of the compiler should not be used while developing programs. Some optimizations move statements from one area of a program to another, or change statements or variables in a way that is not obvious. Since this makes debugging programs more difficult, optimizing before debugging should be avoided. After a program is developed, it can be recompiled with optimization command-line options to improve its performance.

Note: The optimization process is disabled whenever the "g+" option (debugger information) is specified on the command line.

Compiler Directives

The statements in this section are directives to the compiler which appear in the source program. Their effect depends on where they appear in the text. They are not subject to Pascal scope rules as seen in the *RT PC VS Pascal Reference Manual*.

All directives are enhancements to ANSI standards for Pascal. Figure 2-2 lists all the compiler directives in alphabetical order and their functions. Directives may be entered in uppercase or lowercase.

Pascal compiler directives are introduced via toggles embedded in comments. Once an option is selected by a toggle, it remains in effect until

another option in the source code terminates that option. The comment toggle formats are:

```
(*$T parameters*)  
or  
{$T parameters}  
or  
/*$T parameters*/
```

All forms of comment delimiters may be used, and the toggle must immediately follow the opening comment delimiter with no intervening spaces.

The toggle letter, which may be in uppercase or lowercase, is followed by the parameters for that toggle. Compiler directives that are followed by a + or a - may be given in a list, as shown in the following example:

```
{$C+, $I+, $L- }
```

Certain compiler directives can be turned on and off in the source code. In this instance, only the source code between the directives is affected. The format for turning compiler directives on and off is:

```
% "directive" on  
and  
% "directive" off
```

No spaces are allowed after the commas in the list. Any incorrect syntax terminates the scanning of a compiler directive list.

\$C+

CODE GENERATION

turns code generation on. Code generation is done on a procedure-by-procedure basis.

%check off	NO RANGE CHECKING turns off range checking.
%check on	RANGE CHECKING turns on range checking. Range checking is done in assignment statements, on array indexes, and for string value parameters.
	<i>Note:</i> Range checking is done only for user-defined sub-range and scalars, and for array indexing. The integer data type is <i>never</i> range checked. Compile-time range checking (such as <code>v := constant</code>) for user-defined types is always enforced.
%cpage n	PAGE EJECT ejects a page if less than a specified number of lines is left on the current page of the output listing when used in conjunction with the <code>l+</code> or <code>lfilename</code> . This is useful to make sure there is sufficient room for a unit of code, and it does not have to be separated on two different pages. An example of this statement is: <code>%cpage 30</code>
\$D+	DISASSEMBLER INFORMATION turns on the generation of information for the IBM RT PC Disassembler.
\$D-	NO DISASSEMBLER INFORMATION turns off the generation of information for the IBM RT PC Disassembler.
\$E filename	ERROR FILE lists errors to the file specified by <i>filename</i> . To use this directive, the <code>\$L filename</code> directive or the <code>lfilename</code> command-line option must also be specified.
\$F+	FLOATING-POINT HARDWARE generates code to use floating-point hardware.

Note: This feature is similar to the `f+` command-line option described in "Command-Line Options."

\$G+	DEBUG turns on generation of debug information for the RT PC Debugger (sdb).
\$G-	NO DEBUG turns off generation of debug information for the RT PC Debugger (sdb).
\$I-	NO INPUT/OUTPUT CHECKS turns off automatic input/output checks.
\$I <i>filename</i>	INCLUDE SPECIFIED FILE includes the file specified by <i>filename</i> at this point in the source code.
%include <i>filename</i>	INCLUDE FILE inserts source code from a file into the input stream immediately after the current line. The compiler is directed to begin reading its input from a file. When the end of the file is reached, the compiler resumes reading from the original source.
\$L+	LISTING turns listing on without changing the listing file name. The listing file name must be specified before turning the listing on. When the list option is on, the listing is directed to whatever list file was specified on the Pascal compiler's command line. <i>Note:</i> This is the default setting when a listing file is specified on the compiler's command line.
\$L-	NO LISTING turns listing off without changing the listing file name. <i>Note:</i> This is the default setting when a listing file is not specified on the compiler's command line.

\$L filename	<p>LISTING FILE makes a compilation listing in the file specified by the <i>filename</i>. If a listing file already exists, that file is closed and saved before the new file is opened.</p> <p><i>Note:</i> This feature is also available through the <i>lfilename</i> command-line option.</p>
%margins m n	<p>MARGINS redefines the left and right margins of the compiler unit. The <i>m</i> specifies the new left margin, and the <i>n</i> specifies the new right margin. The compiler ignores all characters that lie outside the margins.</p> <p>If this statement appears in a file which is being inserted by the %include statement, the new margins are effective only while the file's source code is being compiled. When the end of the file is reached and the previous source is resumed, the margin settings revert to their previous definitions.</p> <pre>%margins 10 80</pre>
\$N+	<p>CHECK FLOATING-POINT EXPRESSIONS checks the results of floating-point expressions for validity. This option checks the value of most floating-point expressions for Not-a-Number (NaN) and infinity, and generates an error message if either is found.</p>
%page	<p>NEW PAGE forces a skip to the next page on the output listing of the source program.</p>
%print off	<p>NO SOURCE PRINTED turns off the printing of the source code in the listing.</p>
%print on	<p>SOURCE PRINTED turns on the printing of the source code in the listing.</p>

\$Q-	DISPLAY INFORMATION ON COMPILE PROGRESS instructs the Pascal compiler to display more information on the progress of the compile.
\$R+	RANGE CHECKING turns run-time range checking on. <i>Note:</i> Range checking is done only for user-defined sub-range and scalars, and for array indexing. The integer data type is <i>never</i> range checked. Compile-time range checking (such as <code>v := constant</code>) for user-defined types is always enforced.
\$S <i>segment</i>	PLACE CODE MODULES IN SEGMENT places code modules in the segment specified by <i>segment</i> . The default segment name is ' ' (eight spaces), which is where the main program and all built-in support code are always linked. All other code can be placed in any segment. Under the AIX Operating System, segmentation is automatically done by the system and there is no need to explicitly segment programs.
%skip <i>n</i>	SKIP LINES inserts one or more blank lines into the source listing. %skip 2
%title <i>character string</i>	TITLE IN LISTING sets the title in the listing. It also causes a page skip. The title is printed as specified in the statement, that is, there is no change from lowercase to uppercase.
\$U <i>filename</i>	SEARCH FOR UNITS searches for subsequent units in the file specified by <i>filename</i> .
\$%+	PERCENT SIGN IS VALID CHARACTER specifies that the percent sign (%) in identifiers is valid as a character.

Directive	Function
\$C+	Code generation
%check off	No range checking
%check on	Range checking
%cpage	Page eject
\$D+	Disassembler information
\$D-	No disassembler information
\$E <i>filename</i>	Error file
\$F+	Floating-point hardware
\$G+	Debug
\$G-	No debug
\$I-	No input/output checks
\$I <i>filename</i>	Includes specified file
%include	Includes specified file
\$L+	Listing
\$L-	No listing
\$L <i>filename</i>	Listing file
%margins	Margins
\$N+	Checks floating-point expressions
%page	New page

Figure 2-2 (Part 1 of 2). Compiler Directives

Directive	Function
%print off	No source printed
%print on	Source printed
\$Q-	Displays information on compile progress
\$R+	Range checking
\$S <i>segment</i>	Places code modules in segment
%skip <i>n</i>	Skip lines
%title	Title in listing
\$U <i>filename</i>	Searches for units
\$%+	Percent sign is valid character

Figure 2-2 (Part 2 of 2). Compiler Directives

Chapter 3. Using Input and Output Facilities

This chapter provides information on using the RT PC VS Pascal input and output facilities under the AIX Operating System in these areas:

- Opening Files for Input and Output
- Open Options
- Terminal Input and Output

For a description of the available Pascal input/output facilities, see the *RT PC VS Pascal Reference Manual*.

Opening Files for Input and Output

The name of a file to be opened for input or output may be environment-determined or program-determined. Using an environment-determined file name permits the file name to be explicitly designated on an AIX command line at program execution time. This is done through the facilities of AIX environment variables. File names can also be determined from within programs. Program-determined file names are resolved using the options described in "Program-Determined Files."

Environment-Determined Files

The name of an input or output file can be determined at program execution time by using environment-determined file names. The environment variable file names are specified by using a program variable as the name of a file. This permits access by a different file each time the program is executed. The two methods for opening environment-determined files are:

- using environment variables on the command line
- using environment variables in shell scripts

The Pascal program, "mypgm", is a sample program used in the descriptions of the input and output procedures throughout this chapter:

```
program mypgm;
var
    infile,
    outfile  :  text;
    buffer   :  string(1000);
begin
    reset(infile);
    rewrite(outfile);
    while not eof(infile) do
    begin
        readln(infile, buffer);
        writeln(outfile, buffer);
    end;
end.
```

Using Environment Variables on the Command Line

AIX environment variables are used to associate a file name with the program variable that is chosen (for example, "infile" in the program "mypgm"). The following AIX commands illustrate the use of environment variables in the Bourne shell, the C shell, and the DOS shell. These commands are entered on the command line prior to the invocation of the program:

Bourne shell

```
ENVIRONMENT-NAME=file-name; export ENVIRONMENT-NAME
```

C shell

```
setenv ENVIRONMENT-NAME file-name
```

DOS shell

```
set ENVIRONMENT-NAME=file-name
```

ENVIRONMENT-NAME

is the same as the file variable name being used.

Note: Environment variables under the AIX Operating System are case sensitive.

file-name

is the actual file name used in the AIX file system.

Note: Unless otherwise specified, the examples in this chapter use the Bourne shell. For a detailed description of the C shell and the DOS shell, see the *Using and Managing the AIX Operating System* and the *AIX Operating System DOS Services Reference* manuals.

In the following example, the environment variable "INFILE", which is used as a program variable, is associated with the file "file1.in.a1". The program "mypgm" is then executed.

```
INFILE=file1.in.a1; export INFILE  
mypgm
```

After execution, the exported filename, "INFILE", remains in the AIX environment for any subsequent executions which use the same variable and AIX file. To run the same program using a different file, the new file name must be associated with the "INFILE" environment variable and exported again.

It is possible to execute one program in the background and to execute the same program in the foreground using different files, as shown in this example:

```
INFILE=file1.in.a1; export INFILE; mypgm&
```

```
INFILE=file2.in.a1; export INFILE  
mypgm
```

By entering the program invocation on the same line as the environment statements, you associate each line's statements with its own unique AIX process. Environment variables are only known in their current environment. Therefore, "file1" is local to the first invocation of "mypgm", and "file2" is local to the second invocation of "mypgm".

Command-Line Options

Command-line options may be used when associating an environment variable with a file name and are identified by enclosing the contents of the environment variable in parentheses.

The forms of commands using command-line options are:

Bourne shell

```
ENVIRONMENT-NAME='(option,...)'
```

C shell

```
setenvENVIRONMENT-NAME '(option,...)'
```

Note: Options may not be used when setting an environment variable in the DOS shell. The environment variable must be defined in another shell and exported before the DOS shell is initiated.

The available command-line options are:

name=string

specifies that *string* is the name of the AIX file associated with the environment variable. This option must always be included when command-line options are used or the file name is undetermined. If the *string* portion of option is not included, an error occurs.

Bourne shell example:

```
INFILE='(name=file1.in.a1)'; export INFILE
```

C shell example:

```
setenv INFILE '(name=file1.in.a1)'
```

disp=mod

appends records to an existing file. If a file's environment variable is defined with the **disp=mod** option and **rewrite** is used to open the file, output is appended to the end of the existing file. If the file does not exist, it is created.

Bourne shell:

```
INFILE='(name=file1.in.a1, disp=mod)'; export INFILE
```

C shell:

```
setenv INFILE '(name=file1.in.a1, disp=mod)'
```

Note: Options are not case sensitive and may be given in uppercase or lowercase.

Using Environment Variables in Shell Scripts

The easiest and most efficient method to execute a program containing a variable name involves the use of a shell script. All the necessary commands can be put into the shell script. When the name of the shell script file is invoked, each command in the shell script file is executed sequentially. For a complete description of shell script usage, see the manual *Using and Managing the AIX Operating System*.

The shell script allows the use of the association of environment variable with the same file name each time, or with a different file name each time.

Shell Script Using the Same File Name

The following example illustrates a shell script called "run1" that associates the environment variable named "INFILE" with the file named "file1.in.a1". It also contains the command to execute the program "mypgm".

The shell script "run1" contains:

```
INFILE=file1.in.a1; export INFILE
mypgm
```

After the shell script is created as a file and is made executable through the command `chmod 755 run1`, it can be executed by entering:

```
run1
```

When "run1" is executed, it is considered an AIX process. Therefore, anything which executes within the shell script is a child of that process. Since a child process is only known to its parent, the content of the "INFILE" environment variable is local to the "run1" shell script and unknown to other AIX processes.

Shell Script Using Different Files

To prevent having to edit the shell script whenever a different AIX file is used, the shell script can be created with a variable in place of the file name. Using the same shell script shown in the previous example, a variable "\$1" is used in place of the physical filename.

```
INFILE=$1; export INFILE  
mypgm
```

To execute the shell script using the "file1.in.a1" file, enter:

```
run1 file1.in.a1
```

When the shell script is executed, "\$1" is replaced with "file1.in.a1". This allows "run1" to be executed using different file names. It also allows the execution of "run1" to proceed in the background under one name and in the foreground under a second name. For example:

```
run1 file1.in.a1 &  
run1 file2.in.a1
```

Program-Determined Files

AIX file names can also be determined from within programs which is done using open options. This method lets you control which file is being opened.

Open Options

All Pascal procedures that open files are defined with an optional string expression that contains options pertaining to the file being opened. These options determine how the file is opened and its attributes.

The format for using open options is the same for **reset**, **rewrite**, and **update**.
The format for **reset** is:

reset (*f*)
or
reset (*f*, *options*)

f
is a file variable.

options
is a string expression or constant that contains one or more open options. Options must be separated by commas. The available options are:

DDNAME=*name*
specifies that the AIX file associated with the file variable is in the environment variable indicated by *name*. The option name must be entered in uppercase letters.

An example of the **DDNAME** option is:

```
reset (infile, 'DDNAME=infile2')
```

The environment variable then needs to be defined. The command line options described in this chapter may be used when defining the environment variable. The following statements illustrate different ways of defining the environment variable "infile2":

```
infile2=file1.in.a1; export infile2  
infile2='(name=file1.in.a1)'; export infile2
```

If this option is not specified, the environment variable name associated with the file is derived according to these rules:

- If the file variable is a simple variable, the default environment variable name is the name of the file variable itself.

- If the file variable is a pointer qualified (that is, a pointer used in any part of the file variable), an element of an array, or a field of a record, the default environment variable name is **pascal nnn** where nnn is a number. For example, if "infile" is defined as a field of a record and a `reset (infile);` procedure is used, the default environment variable is "pascal1". The second file opened by **reset**, **rewrite**, or **update** is associated with the environment variable "pascal2".

The **DDNAME** option is applicable to the **reset**, **rewrite**, and **update** procedures.

INTERACTIVE

indicates that the file is opened for input as an interactive file.

This option applies to the **reset** procedure only and is implied in the **termin** procedure.

Note: For files to be interactive, each **reset** statement must contain the **INTERACTIVE** option. If the **reset** statement does not contain **INTERACTIVE**, the files are no longer interactive files.

NAME=file name

specifies the name of an AIX file that is opened and associated with the file variable.

Note: The **NAME** option takes precedence over any environment variable assignments or **DDNAME** assignments.

The **NAME** option is applicable to the **reset**, **rewrite**, and **update** procedures.

UCASE

causes the text read from a file opened by **termin** to be translated to uppercase.

Examples:

These are valid option string expressions:

```
'NAME=file1.in.a1'  
'NAME' || var1  
var2
```

```
var1  
  contains =file1.in.a1
```

```
var2  
  contains NAME=file1.in.a1
```

Terminal Input and Output

The processes for using files for interactive processing with the **reset**, the **termin**, and **termout** procedures are described in this section. No initial input/output operation is performed on files opened interactively until a **read** statement is encountered. These processes also force the **eof** function to return the value "false".

Both **reset** and **termin** default to the standard input device. To set **eof** to "true" for an interactive input file, the **eof** character must be entered. The default **eof** character is Ctrl-D. For a detailed description of the **eof** character and standard input, see the *Using and Managing the AIX Operating System* manual.

Using Reset

To indicate that a file is opened for interactive input, specify "INTERACTIVE" in the options string of the **reset** procedure.

Example:

```
reset (infile, 'INTERACTIVE');
```

Using Termin and Termout

The **termin** procedure opens a text file for interactive input from the keyboard, and the **termout** procedure opens a text file for terminal output.

Examples:

```
termin (infile, 'UCASE');  
termout (outfile);
```

Note: If several calls to **termin** are made in a single program, the option used with the last call to **UCASE** determines whether standard input is in uppercase or lowercase for all files opened by **termin** and normal standard input. If the **termin** statement does not contain **UCASE**, the text being read from the file is read in lowercase.

Chapter 4. Data Representations

This chapter describes the ways in which IBM RT PC VS Pascal represents data in storage. It is intended as a guide for writing modules in languages other than Pascal and having those modules interface to Pascal.

Storage Allocation

This section describes the way in which storage is allocated to variables of various types. In general, a halfword value (a value that occupies 16 bits) is aligned on a halfword boundary, and values larger than a halfword are aligned on a word boundary. A word boundary is a value that occupies 32 bits. Values that can fit into a single byte (8 bits) are aligned on a byte boundary.

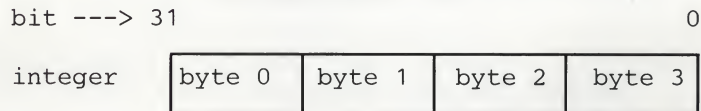
boolean variable	occupies 1 byte of storage aligned on a byte boundary. A value of 0 represents the value "false". A value of 1 represents the value "true". Any other value is an undefined boolean value.
scalar (ordinal) type	occupies 1 byte of storage aligned on a byte boundary for a type of 128 or less elements. If there are more than 128 elements in the scalar type, it occupies a halfword. A scalar type is assigned the values 0, 1, 2 to $n-1$, where n is the number of elements in the scalar.
subrange element	occupies 1 byte aligned on a byte boundary when in the range -128 to 127. A subrange element in the range -32768 to 32767 occupies a halfword aligned on a halfword boundary. A subrange element

	greater than that occupies 1 word aligned on a word boundary.
unpacked char	occupies 1 word.
integer	occupies 1 word aligned on a word boundary.
shortreal	occupies 1 word aligned on a word boundary. All shortreal arithmetic is done using single precision floating point instructions. The data is stored according to the ANSI/IEEE 770X3.97-1983 standard for floating point data. The range of shortreal numbers is approximately $-3.4\text{E}-38$ to $3.4\text{E}38$ with a precision of approximately seven decimal places. Normal arithmetic operations upon shortreal data types can result in the extreme values of +infinity, -infinity, or Not-a-Number (NaN).
real	occupies 2 words aligned on a word boundary. All real arithmetic is done using double precision floating-point. The data is stored according to the ANSI/IEEE 770X3.97-1983 standard for floating point data. The range of real numbers is approximately $-1.8\text{E}-308$ to $1.8\text{E}308$ with a precision of approximately 15 decimal places. Normal arithmetic operations upon real data types can result in the extreme values of +infinity, -infinity, or Not-a-Number (NaN).

Whatever the size of the data element in question, the most significant bit of the data element is always in the lowest-numbered byte of however many bytes are required to represent that object.

Representation of Integers

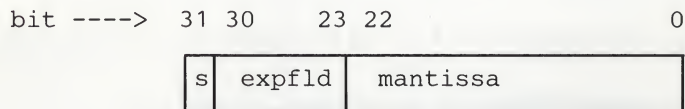
Integer data types are illustrated in the following diagram:



Representation of Shortreals

Shortreal data elements are stored according to the ANSI/IEEE 770X3.97-1983 standard for floating-point data and are illustrated in the following diagrams.

Shortreal (Single-Precision Floating-Point)



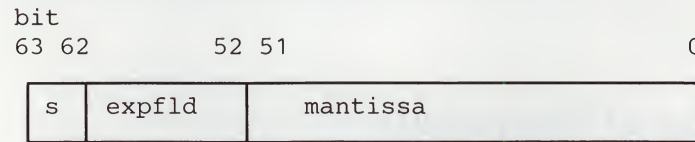
The three fields of a shortreal, or single-precision floating-point number are:

- a sign bit designated by "s" in the diagram. The sign bit has the value "1" only if the number is negative.
- an 8-bit biased exponent. The values of all ones and all zeros are reserved values for the exponent.
- a 24-bit mantissa with the high-order 1-bit "hidden".

Representation of Reals

Real data elements are stored according to ANSI/IEEE standard and are illustrated in the following diagram:

Real (Double-Precision Floating-Point)



The three fields of a real or double-precision floating-point number are:

- a sign bit designated by "s" in the diagram. The sign bit has the value "1" only if the number is negative.
- an 11-bit biased exponent. The values of all zeros and all ones are reserved values for the exponent.
- a normalized 53-bit mantissa with the high-order 1-bit "hidden".

A real number is represented by the form:

$$2^{\text{exponent-bias}} * 1.f$$

The *f* is the number of bits in the mantissa.

Representation of Extreme Values

This section describes how to represent "values" such as +infinity and -infinity when real data elements are stored in the system, and describes their behavior in the evaluation of an expression.

Zero (signed) is represented by an exponent of zero and a mantissa of zero.

Denormalized numbers are a product of "gradual underflow" and are nonzero numbers with an exponent of zero. The form of a denormalized number is:

$$2^{\text{exponent-bias}+1} * 0.f$$

The f is the number of bits in the mantissa.

Signed infinity (that is, affine infinity) is represented by the largest value that the exponent can assume (all ones), and a nonzero mantissa. The sign is usually ignored.

Normalized real numbers are said to contain a "hidden" bit, which provides one more bit of precision than would otherwise be the case.

Hexadecimal Representation of Selected Numbers

The hexadecimal representation of selected numbers is:

Value	Real
+0	00000000
-0	80000000
+1.0	3F800000
-1.0	BF800000
+2.0	40000000
+3.0	40400000
+Infinity	7F800000
-Infinity	FF800000
NaN	7F8xxxxx

Figure 4-1. Hexadecimal Representation

The extreme real values are printed as follows:

+infinity	prints as a row of plus signs.(+)
-infinity	prints as a row of minus signs.(-)
NaN (Not-a-Number)	prints as a row of question marks.(?)

The number of characters printed equals the field width.

Representation of Sets

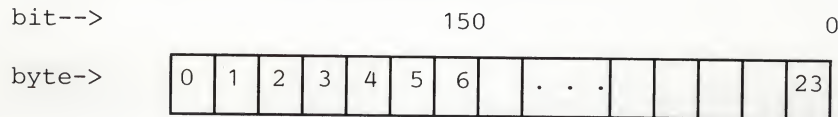
RT PC VS Pascal represents a set like a "giant integer". The "zero'th" element of a set is always present in the set. For example, suppose that a type and a variable are defined like this:

```

type
    settype = 1..150;
var
    aset: settype;

```

The representation for the variable *aset* is then:



The number of bytes required to contain this set of 1 to 150 is $150/8$, which is 18 bytes. Since set storage is allocated in 4-byte increments, the number would become 20 bytes with 2 unfilled bytes. When the set is stored in memory, a 4-byte length field is added to the beginning of the set. The entire length of the set including the concatenated 4 bytes is stored in the length field. In the diagram, the length field would contain a value of 24.

The storage is allocated as shown in the diagram. The value $150 \bmod 8$ is 6, and there are 2 unfilled bits in the least significant byte (byte 6) of the set. In the above example, the length is actually 24 (18 bytes, 4 concatenated to the beginning, 2 unfilled) with the leftmost 2 "real" bytes of the set unfilled (bytes 4 and 5). The location of bit 5, the 5th entry in the set, would be the 5th bit from the right in byte 23.

Representation of Arrays

Components of unpacked arrays and records are allocated contiguously. There is no attempt made to conserve space in units smaller than bytes.

Arrays are stored in row order; that is, the last index varies fastest. This follows from the strict definition of a multidimensional array in Pascal:

```

array[first index] of array[second index].. of
    array[n'th index] of whatever type;

```

Representation of Pointers

Pointers always occupy 4 bytes. The nil pointer is represented by a value of zero.

Representation of Strings

The string data type has a 4-byte word-aligned string length followed by a set of contiguous bytes, one character per byte. The dynamic length of the string can be determined using the **length** function.

Packing Methods

The "packed" data type attribute, when used in IBM mode, has no affect on storage allocation.

Chapter 5. Mixing Languages

The RT PC language system permits the mixing elements from different languages in a single program. This chapter assumes you are familiar with the languages you wish to mix; the elements of the languages are not described here in detail.

Note: In this chapter, the FORTRAN language described is IBM RT PC VS FORTRAN; the Pascal language is IBM RT PC VS Pascal; the C language is IBM RT PC C.

Correspondence of Data Types

The data types of one language are usually quite different from the data types of another language. Also, the way data is stored is not the same across languages; the internal data representation is left unspecified and usually varies with the implementation.

However, a certain amount of similarity among the data types of the different languages exists since the languages share many system primitives and since IEEE standard data representations are used as much as possible. Figure 5-1 shows some of the correspondence among languages.

Note: Figure 5-1 shows how the languages represent data internally in the computer's memory rather than how data are passed between program units.

FORTRAN IBM and R1 Modes	FORTRAN VX Mode	Pascal	C
LOGICAL*1	LOGICAL*1	BOOLEAN	
	LOGICAL*2		
LOGICAL*4	LOGICAL*4		
INTEGER*2	INTEGER*2		short
INTEGER*4	INTEGER*4	INTEGER	int
REAL*4	REAL*4	SHORTREAL	float
REAL*8	REAL*8	REAL	double
COMPLEX	COMPLEX		
COMPLEX*8	COMPLEX*8		
COMPLEX*16	COMPLEX*16		
CHARACTER	CHARACTER	packed array of CHAR	char
		STRING	

Figure 5-1. Correspondences of Data Types Among RT PC VS Languages

As Figure 5-1 shows, each language has data types that do not exist in the other languages. When you interface languages, make sure you either avoid mismatching data types or use the mismatches very cautiously. When data types do correspond, the interfacing of the languages is very straightforward.

Most numeric data types have counterparts across the languages. However, character and string data types do not. The most difficult aspect of language interfacing is the passing of character, string, or text variables between languages.

FORTRAN's only character variable type is CHARACTER, which is stored as a set of contiguous bytes, one character per byte. The length of a FORTRAN character variable or character array element is determined at

compile time and is therefore static. Character lengths are returned by the FORTRAN intrinsic function LEN.

Pascal's character-variable data types are STRING and packed array of CHAR. The STRING data type has a 4-byte word-aligned string length followed by a set of contiguous bytes, one character per byte. The dynamic length of the string can be determined using the **length** function. Packed array of CHAR, however, like FORTRAN's CHARACTER type, is stored as a set of contiguous bytes, one character per byte.

C character data is typically stored as arrays of type "char". The "char" data type stores one character per byte; therefore, an array of "char" is stored exactly like a FORTRAN CHARACTER variable or a Pascal packed array of CHAR.

Storage of Matrices

FORTRAN matrices are stored in computer memory by column (column major order); therefore, the first subscript in a multi-dimensional array varies fastest. An array dimensioned as $A(3,2)$ is stored in this order: $A(1,1)$, $A(2,1)$, $A(3,1)$, $A(1,2)$, $A(2,2)$, and $A(3,2)$.

Pascal and C matrices are stored in computer memory by row (row major order). For example, if an array in Pascal is declared as `A : array [1..3, 1..2] of REAL`, it is stored in this order: $A[1,1]$, $A[1,2]$, $A[2,1]$, $A[2,2]$, $A[3,1]$, and $A[3,2]$.

Since the matrix storage convention for Pascal and C differs from that for FORTRAN, be careful when passing references to matrices between FORTRAN and the other languages.

Input/Output Primitives

Primitive input/output routines are usually bound to a user's program during the final linking process when the AIX linker includes the necessary code from the language run-time library ("`libvsfor.a`" for FORTRAN), and the system run-time library ("`libvssys.a`" for FORTRAN and Pascal).

When you mix, for example, FORTRAN and Pascal, you must remember to link both "`libvsfor.a`" and "`libvssys.a`" in this order. This allows the primitives needed for both the FORTRAN and Pascal parts of the program to be present.

The input/output primitives are different for each language; because of this you are not able, for example, to open a file or device for use by one language and write to it or read from it in a different language. Generally, however, two languages can exist in a program as long as they each have their own files and device for input/output, which includes the console device.

Calling from a Non-RT PC VS Main Program: When the main program is compiled using a non-RT PC VS compiler, special handling is required when calling RT PC VS Pascal and FORTRAN subroutines that perform input/output.

An initialization routine must first be called from the main program so that the input/output buffers and information are properly set up: for RT PC VS Pascal, the routine is "`vs__pio`" and is in the system run-time library "`libvssys.a`"; for RT PC VS FORTRAN, the routine is "`vs__fio__`" and is in the language run-time library "`libvsfor.a`". These routines do not require parameters.

Note: When using RT PC FORTRAN languages, the trailing underscore is automatically appended to the routine name; therefore the name to be coded is "`vs__fio`".

When the main program is compiled using RT PC VS Pascal or FORTRAN but the subroutine to be called is not, you need to first become familiar with the requirements of that particular subroutine.

Subroutine Linkage Convention

The "subroutine linkage convention" describes the machine state at subroutine entry and exit. This scheme allows routines that are compiled separately in the same or a different RT PC language to be linked and executed when called.

Load Module Format

The load module format used is AIX GPOFF (General Purpose Output File Format). For the GPOFF, each routine has a "constant pool" in the data segment. A constant pool is a data area created for each routine. The first word of each routine's constant pool contains the address of the routine's entry point. A constant pool also provides the routine with addressability to constants, local data, and any called-routine's constant pool. A constant pool pointer (cpp) is passed in register 0 on a call.

Register Usage

If a register is not saved during the call, its contents may be changed during the call. Conversely, if a register is saved, its contents are not changed, and the register can be used as "scratch" (that is, as a work area). Figure 5-2 lists registers and their functions.

Register	Name	Saved During Call	Use
0	called cpp	no	Constant pool pointer. On call, contains address of called routine's constant pool. Can also be used for scratch between calls.
1	fp	yes	Stack pointer
2	--	no	On call, first word of parameter words to called routines. On return, first word of return value. Between calls, can be used as scratch.
3	--	no	On call, second word of parameter words to called routines. On return, second word of return value (for example, low-order 2 words of a floating-point value). Between calls, can be used as scratch.
4	--	no	On call, third word of parameter words to called routines. Between calls, can be used as scratch.
5	--	no	On call, fourth word of parameter words to called routines. Between calls, can be used as scratch.
6	--	yes	Not involved in call interface. Can contain register variables or can be used as scratch.

Figure 5-2 (Part 1 of 2). Register Usage

Register	Name	Saved During Call	Use
7	--	yes	Not involved in call interface. Can contain register variables or can be used as scratch.
8	--	yes	Not involved in call interface. Can contain register variables or can be used as scratch.
9	--	yes	Not involved in call interface. Can contain register variables or can be used as scratch.
10	--	yes	Not involved in call interface. Can contain register variables or can be used as scratch.
11	--	yes	Not involved in call interface. Can contain register variables or can be used as scratch.
12	--	yes	Not involved in call interface. Can contain register variables or can be used as scratch.
13	--	yes	Frame pointer
14	current cpp	yes	Not involved in call interface. By convention, however, contains address of current routine's constant pool.
15	link	no	On call, contains return address. Can also be used as scratch.

Figure 5-2 (Part 2 of 2). Register Usage

Stack Frame

When a routine is called, the compiler passes parameter words 5 through n onto the stack. Space is allocated for parameter words 1 through 4. If the routine uses local or temporary variables, they are allocated space on the stack. The stack grows from higher addresses to lower addresses. A single frame-pointer register (register 13) is used to address local storage, incoming and outgoing parameters, and the save area.

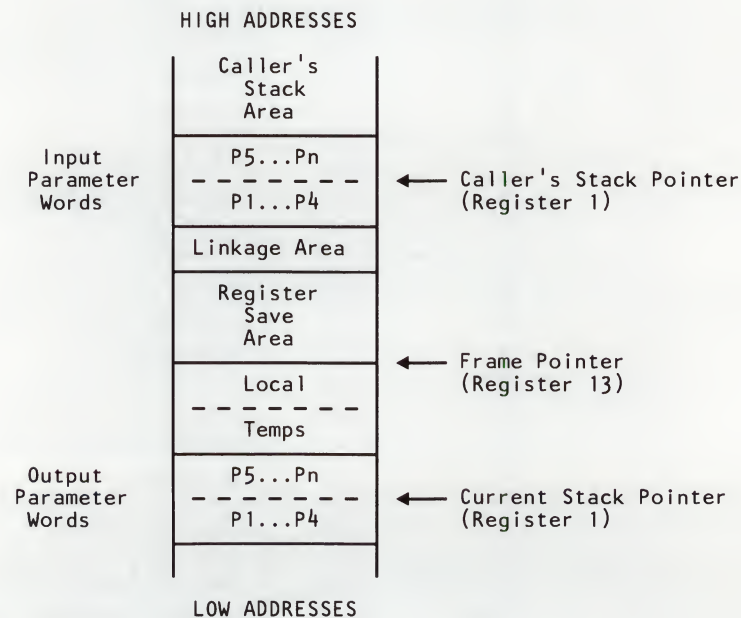


Figure 5-3. Contents of a Stack Frame

Figure 5-3 represents the contents of a stack frame. The areas in the stack are described as follows:

- Input parameter words

- Linkage area
- Register save area
- Local and temporary stack area
- Frame pointer
- Output parameter words
- Total stack frame

Input Parameter Words: If a routine receives more than 4 parameter words, the stack pointer (register 1) upon entry addresses the locations in the stack where parameter words 5 through n are stored. Immediately below the stack pointer is a 4-word area in which the first 4 parameter words (passed in registers 2 through 5) can be stored by the compiler. The parameter words are stored only if registers 2 through 5 are to be used as scratch registers or if a parameter address is required. This area is present regardless of the number of parameters being passed.

```
regparsize = 16      # Size of area in which first 4
                     # parameter words can be stored.
```

Linkage Area: The first word of the linkage area is reserved for storing the environment pointer, which is the frame pointer (register 13) of the routine in which the current routine is nested. It is used to gain addressability to the enclosing routine's local variables. Internal calls to nested routines do not use this pointer. It is required to support Pascal parametric procedures or functions since they may be called from a separately compiled routine.

```
envirsize = 4        # Environment pointer size
```

The next 4 words of the linkage area are reserved.

```
resrvsize = 16      # Reserved area size
linksize = envirsize + # Link area size
           resrvsize
```


Register Save Area: The general-purpose registers (GPRs) and floating-point registers (FPRs) are saved in the register save area. GPR 15 is always saved in the highest word of the register save area. Floating-point registers are saved immediately following the GPRs.

```
Rn                                # First GPR saved (6 <= Rn <= 15)
GPRsize = 4*(16-Rn)              # GPR save area size
save = regargsize +              # Offset of GPR save area
      linksize + GRPsize
FPRsize = 4*163                  # FPR save area size
savesize = GRPsize +             # Total register save area
      FPRsize
```

Local and Temporary Stack Area: When a routine needs space for local or temporary variables, the compiler allocates space for them in the local and temporary stack area. The size of this area is known at compile time.

```
set localsize      # Size of local auto's and temp's
```

Frame Pointer: The compiler uses register 13 as the frame pointer to address sections in the stack frame. The register save area, linkage area, and input parameter words are referenced as positive offsets to register 13. The local and temporary variables are referenced as negative offsets to register 13. Output parameter words are referenced using register 1, the current stack pointer.

Output Parameter Words: If a routine makes a call with more than 4 parameter words, the compiler allocates space for the parameter extension list immediately above the stack pointer. This area is large enough to hold the biggest parameter extension list for any call made by the routine.

```
extlistsize      # Size of biggest parameter extension list
```

Total Stack Frame: The entire stack frame can be thought of as including all the space between the caller's stack pointer and the current stack pointer. It is also reasonable to consider the input parameter area as being part of the current stack frame. In a sense, each parameter area belongs to both the caller's stack frame and the current stack frame. In either case, the stack frame size is best defined as the difference between the caller's stack pointer and the current stack pointer.

```
framesize = regargsize +           # stack frame size
            linksize + savesize +
            localsize + extlistsize
```

Parameter Passing

The contents of the parameter words vary among languages. Parameters are understood to occupy an array in the stack, with each parameter aligned on a word boundary. The compiler allocates space in the stack for all the parameter words, but it does not store the first 4 words on the stack. These values are passed in registers 2 through 5. They are only copied to the stack space if a parameter address is required or if registers 2 through 5 are to be used as scratch registers.

Parameter values are passed according to type:

- A type value less than or equal to 4 bytes is passed right-justified in a single word or register, word aligned.
- A procedure or function parameter is passed as a pointer to the routine's constant pool. The routine's environment pointer is also passed.
- A double value is passed in two successive words, which need not be doubleword aligned. One may be in register 5 and the other in the stack frame.

Function Values

Functions return their values according to type:

- A type value less than or equal to 4 bytes is returned right-justified in register 2.
- A double value is returned in registers 2 and 3.

Parameter Addressing

The input parameter words 5 through n can be addressed in the stack by:

```
linksize + savesize+4*k-4(r13)    # get k-th parameter word
```

If the compiler stored the first 4 parameter words (registers 2 through 5) in the stack frame, then they can be addressed the same way.

Traceback

The compiler supports the traceback mechanism, which is required by the AIX Operating System Symbolic Debugger in order to unravel the call/return stack. Each module has a traceback table in the text segment at the end of its code. This table contains information about the module including the type of module as well as stack frame and register information.

Entry and Exit Code

The compiler adds entry and exit code around each routine's code, which sets up and removes the routine's stack frame.

The entry code:

- saves modified non-volatile registers
- decreases stack pointer (register 1) by `framesize`
- copies the constant pool pointer from register 0 to register 14
- sets frame pointer (register 13); if the routine is the main program, register 13 points to the global data area.

The exit code:

- restores stack pointer (register 1)
- restores registers.

Calling a Routine

A routine has two symbols associated with it: a constant pool pointer (*_name*) and an entry point (*.name*). When a call is made to a routine, the compiler branches to the *.name* entry point directly and loads the *_name* constant pool pointer into register 0. If the routine entry point is not within a megabyte of the call, the compiler loads the *_name* constant pool pointer, loads the *.name* entry point from the first word of the constant pool, and branches to it.

Sample Programs

The following sample programs show ways to connect program units written in different languages. They also illustrate the mechanisms for passing character, integer, and floating-point variables between Pascal, FORTRAN, and C.

In covering these three variable types, an example of each language calling the other languages is given. The sample programs are included only to illustrate the mixing of the languages and do not show all types of parameter passing.

The lists of AIX commands needed to run the sample programs illustrate that the FORTRAN and Pascal source files must be compiled to ".o" files and then linked together (in these cases, along with C files) to produce an executable file. This can be done by executing each pass of the compiler separately and stopping when the ".o" file is produced, or by interrupting the shell script before the link ("cc") step.

Pascal Calling FORTRAN and C

This example illustrates the passing of Pascal CHAR, INTEGER, REAL, and DOUBLE data to a FORTRAN subroutine and a C function. The source code for each program unit and the AIX commands needed to run the program are shown, as well as a sample run of the program.

Note: The FORTRAN compiler appends an underscore (_) to external symbols. Thus the Pascal external declaration must have an underscore appended to the FORTRAN name.

The Calling Pascal Program

```
{ This code is in the file named "pasexam.pas". }

program MAIN (input,output);
  type TEXT = packed array [0..79] of CHAR;

  { Note underscore in FORTRAN procedure declaration }
  procedure FSUB_ (var NAMES : TEXT;
                  COUNT      : INTEGER;
                  var IPAS   : INTEGER;
                  var XPAS   : SHORTREAL;
                  var YPAS   : REAL    ); external;

  procedure CSUB (  NAMES : TEXT;
                  var IPAS : INTEGER;
                  var XPAS : REAL;
                  YPAS     : REAL    ); external;

  procedure PSUB;
  var
    CHRPAS : TEXT;
    INTPAS  : INTEGER;
    XREAL   : SHORTREAL;
    YDOUB   : REAL;
    I       : INTEGER;
```

```

begin
  CHRPAS[0] := 'H'; CHRPAS[1] := 'I'; CHRPAS[2] := ' ';
  CHRPAS[3] := 'W'; CHRPAS[4] := 'I'; CHRPAS[5] := 'R';
  CHRPAS[6] := 'T'; CHRPAS[7] := 'H';
  INTPAS    := 50;
  XREAL     := 10.0;
  YDOUB     := 0.0d0;

  writeln;
  writeln ('Before calls:');
  write(' Text: *'); for I := 0 to 7 do write(CHRPAS[I]);
  writeln('*');
  writeln(' INTPAS=',INTPAS,' XREAL=',XREAL,' YDOUB=',YDOUB);
  FSUB_(CHRPAS,20,INTPAS,XREAL,YDOUB);

  writeln;
  writeln ('After FORTRAN call:');
  write(' Text: *'); for I := 0 to 20 do write(CHRPAS[I]);
  writeln('*');
  writeln(' INTPAS=',INTPAS,' XREAL=',XREAL,' YDOUB=',YDOUB);
  CSUB(CHRPAS,INTPAS,XREAL,YDOUB);

  writeln;
  writeln ('After C call:');
  write(' Text: *'); for I := 0 to 4 do write(CHRPAS[I]);
  writeln('*');
  writeln(' INTPAS=',INTPAS,' XREAL=',XREAL,' YDOUB=',YDOUB);
end;

```

```

begin
  writeln('This message is printed at the beginning of MAIN.');
```

PSUB;

```

  writeln('This message is printed at the end of MAIN.')
```

end.

The Called FORTRAN Subroutine

```
C      This is the FORTRAN subroutine to be called by Pascal.
C      This code is in the file named "pcallf.f".
```

```
SUBROUTINE FSUB(CHR,I,X,Y)
CHARACTER*20 CHR
INTEGER I
REAL X
I=LEN(CHR)
CHR='FORTRAN Lives!'
X=X*I
Y=1.0D0
RETURN
END
```

The Called C Function

```
/*      This is the C function to be called by Pascal.      */
/*      This code is in the file named "pcallc.c".          */

int CSUB (word, /* C subprograms are functions, but the */
          i, /* value returned can be ignored. */
          x,
          y )

char word[79];
int *i;
float *x;
double *y;

{
    word[0] = 'h'; /* Arrays in C always use the */
    word[1] = 'i'; /* call-by-reference mechanism. */
    word[2] = ' ';
    word[3] = 'C';
    word[4] = ' ';
```

```

        *i = -3;
        *x = -1.0;
        *y = 1.0;
        return (0); /* A zero is returned which may be */
                    /* ignored if CSUB is treated as a */
                    /* procedure, or used if CSUB is */
    }              /* treated as a function. */

```

Commands and Output

The AIX commands needed to run this sample program are:

```

vspascal pasexam.pas
vpass2 pasexam.i
vpass3 pasexam.obj
vsfort pcallf.f
vpass2 pcallf.i
vpass3 pcallf.obj
cc -o pasexam pasexam.o pcallf.o pcallc.c -lm
   /usr/lib/libvsfor.a /usr/lib/libvssys.a
pasexam

```

The output from running this sample program is:

This message is printed at the beginning of MAIN.

Before calls:

Text: *HI WIRTH*

INTPAS=50 XREAL= 1.00000E+01 YDOUB= 0.000000000000000D+000

After FORTRAN call:

Text: *FORTRAN lives! *

INTPAS=20 XREAL= 2.00000E+02 YDOUB= 1.000000000000000D+000

After C call:

Text= *hi C *

INTPAS=-3 XREAL=-1.00000E+00 YDOUB= 1.000000000000000D+000

This message is printed at the end of MAIN.

FORTTRAN Calling Pascal and C

This example illustrates the passing of FORTRAN CHARACTER, INTEGER, REAL, and DOUBLE PRECISION data to a Pascal procedure and a C function. The source code for each program unit and the AIX commands needed to run the program are shown, as well as a sample run of the program.

Note: The FORTRAN compiler appends an underscore (_) to external symbols. Thus the name of the FORTRAN routine calling Pascal would be "fcallp" but the called Pascal routine must be named "fcallp_" for linkage resolution.

The Calling FORTRAN Program

```
C      This FORTRAN code is in the file named "forexam.f".
C
      PROGRAM EXAMPLE
      CALL MYCHOICE
      WRITE(*,100)
100    FORMAT(' I've safely returned after doing all that!')
      END

      SUBROUTINE MYCHOICE
      INTEGER IFOR
      REAL XFOR
      DOUBLE PRECISION YFOR
      CHARACTER*10 CHRFOR

      EQUIVALENCE (CHRFOR,LETTER)
      CHRFOR='HELLO'
      IFOR=50
      XFOR=10.
C      Some data is initialized.
      YFOR=0.
      WRITE(*,100) CHRFOR,IFOR,XFOR,YFOR
100    FORMAT('/'Before calls: '/' Text string: *'A'*'
+          '/' IFOR='I10/' XFOR='F10.2/' YFOR='F10.2)
```



```

C      A Pascal procedure is called.
      CALL PSUB(CHRFOR,IFOR,XFOR,YFOR)
      WRITE(*,110) CHRFOR,IFOR,XFOR,YFOR
110    FORMAT('/'After Pascal call:/' Text string: *'A'*'
+      /' IFOR='I10/' XFOR='F10.2/' YFOR='F10.2)

C      A C subroutine is called.
      CALL CSUB(CHRFOR,IFOR,XFOR,YFOR)
      WRITE(*,120) CHRFOR,IFOR,XFOR,YFOR
120    FORMAT('/'After C call:/' Text string: *'A'*'
+      /' IFOR='I10/' XFOR='F10.2/' YFOR='F10.2)

      END

```

The Called Pascal Procedure

```

{ This is the Pascal procedure to be called by FORTRAN.
  This code is in the file named "fcallp.pas". }

```

```

segment DUMMYNAME;

```

```

      type TEXT = packed array [0..79] of CHAR;

```

```

      procedure PSUB_(var WORDS : TEXT;
                      COUNT : INTEGER;
                      var I    : INTEGER;
                      var X    : SHORTREAL;
                      var Y    : REAL);external;

```

```

      procedure PSUB_;
      begin
        WORDS[0] := 'B'; WORDS[1] := 'Y'; WORDS [2] := 'E';
        WORDS[3] := ' '; WORDS[4] := ' ';
        X := X * I;
        I := COUNT;
        Y := 1.0d0;
      end;

```

The Called C Function

```
/* This is the C function to be called by FORTRAN. */
/* This code is in the file named "fcallc.c". */

/* Note underscore in procedure declaration. */

int csub_ (word, /* C subprograms are functions, but the */
          count, /* value returned can be ignored. */
          i,
          x,
          y)

char word[79];
int count;
int *i;
float *x;
double *y;

{
    word[0] = 'h'; /* Arrays in C always use the */
    word[1] = 'i'; /* call-by-reference mechanism. */
    word[2] = ' ';
    word[3] = 'C';
    word[4] = ' ';

    *i = -3;
    *x = -(*x);
    *y = 2.0;
    return (0); /* A zero is returned which may */
               /* be ignored. */
}
```

Commands and Output

The AIX commands needed to run this sample program are:

```
vsfort forexam.f
vspass2 forexam.i
vspass3 forexam.obj
vspascal fcallp.pas
vspass2 fcallp.i
vspass3 fcallp.obj
cc -o forexam forexam.o fcallp.o fcallc.c -lm
   /usr/lib/libvsfor.a /usr/lib/libvssys.a
forexam
```

The output from running this sample program is:

Before calls:

```
Text string: *HELLO      *
IFOR=        50
XFOR=       10.00
YFOR=        .00
```

After Pascal call:

```
Text string: *BYE       *
IFOR=        10
XFOR=       500.00
YFOR=        1.00
```

After C call:

```
Text string: *hi C      *
IFOR=        -3
XFOR=      -500.00
YFOR=        2.00
```

I've safely returned after doing all that!

C Calling FORTRAN and Pascal

This example illustrates passing C-language char, int, float, and double data to a FORTRAN subroutine and a Pascal procedure. The source code for each program unit and the AIX commands needed to run the program are shown, as well as a sample run of the program.

Note: The FORTRAN compiler appends an underscore (_) to external symbols. Thus the C external declaration must have an underscore appended to the FORTRAN name.

The Calling C Program

```
/* This code is in the file named "cexam.c". */

#include <stdio.h>
main ()
{
    printf("\n This message is printed at the start of MAIN.");
    cfunc ();
    printf("\n This message is printed at the end of MAIN.");
}

cfunc ()
{
    char chrc[79];
    int ic, count;
    float xc;
    double yc;

    chrc[0] = 'h'; chrc[1] = 'i'; chrc[2] = ' ';
    chrc[3] = 'C'; chrc[4] = '\0';
    ic = 50; xc = 10.; yc = 0.0;
    count=10;

    printf("\n Before calls:");
    printf("\n  Text string: %s", chrc);
    printf("\n   ic = %d", ic);
    printf("\n   xc = %f", xc);
    printf("\n   yc = %f", yc);
```

```

fsub_(chrc,count,&ic,&xc,&yc);      /* Arrays in C always use */
printf("\n After FORTRAN call:"); /* the call-by-reference */
printf("\n  Text string: %s", chrc); /* mechanism.          */
printf("\n  ic = %d", ic);
printf("\n  xc = %f", xc);
printf("\n  yc = %f", yc);

psub(chrc,&ic,&xc,&yc);
printf("\n After Pascal call:");
printf("\n  Text string: %s", chrc);
printf("\n  ic = %d", ic);
printf("\n  xc = %f", xc);
printf("\n  yc = %f", yc);
}

```

The Called FORTRAN Subroutine

```

C      This is the FORTRAN subroutine to be called by C.
C      This code is in the file named "ccallf.f".

      SUBROUTINE FSUB(WORDS,I,X,Y)
C      FORTRAN uppercases all globals. Note that the order of
C      the parameters is the order in the calling program unit.
      CHARACTER*80 WORDS
      INTEGER I
      REAL X
      DOUBLE PRECISION Y
      WORDS='FORTRAN lives! '//Char(0)
C      The string terminator C expects is concatenated.
      I=LOG(X)
      X=LOG(X)
      Y=45.DO
      RETURN
      END

```

The Called Pascal Procedure

```
{ This is the Pascal procedure to be called by C.
  This code is in the file named "ccallp.pas". }

segment DUMMYNAME;

  type TEXT = packed array [0..79] of CHAR;

  procedure PSUB (var WORDS : TEXT;
                  var I      : INTEGER;
                  var X      : SHORTREAL;
                  var Y      : REAL );external;

  procedure PSUB;
  begin
    WORDS[0] := 'G'; WORDS[1] := 'o'; WORDS[2] := ' ' ;
    WORDS[3] := 'W'; WORDS[4] := 'i'; WORDS[5] := 'r' ;
    WORDS[6] := 't'; WORDS[7] := 'h'; WORDS[8] := '!' ;
    WORDS[9] := chr(0); { C character string terminator }
    X := 2*X;
    I := 2*I;
    Y := 4.0d0;
  end;
```

Commands and Output

The AIX commands needed to run this sample program are:

```
vspascal ccallp.pas
vspass2 ccallp.i
vspass3 ccallp.obj
vsfort ccallf.f
vspass2 ccallf.i
vspass3 ccallf.obj
cc -o cexam cexam.c ccallp.o ccallf.o -lm
   /usr/lib/libvsfor.a /usr/lib/libvssys.a
cecam
```


The output from running this sample program is:

This message is printed at the start of main.

Before calls:

Text string: hi C

ic= 50

xc= 10.000000

yc= 0.000000

After FORTRAN call:

Text string: FORTRAN lives!

ic= 2

xc= 2.302585

yc= 45.000000

After Pascal call:

Text string: Go Wirth!

ic= 4

xc= 4.605170

yc= 4.000000

This message is printed at the end of main.

Chapter 6. The Disassembler

The Disassembler produces assembly language listings for Pascal and FORTRAN programs. With the Disassembler, binary code modules created by high-level languages can be translated into assembly language equivalents.

The assembly language output includes:

- absolute address listing
- hex code listing
- variable type listing
- variable location listing
- symbolic references to external entry points
- labels indicating high-level language source-line numbers
- indications of high-level language variable storage locations
- disassembly of certain embedded data constructs used in high-level languages.

The Disassembler is flexible and easy to use, and can be executed in a variety of ways to suit your needs.

Preparation

Before the Disassembler can be used, it is necessary to compile the source program with the "d+" option specified on the command line. This option instructs the RT PC VS Pascal and RT PC VS FORTRAN compilers to place additional tables of symbolic information into the binary code, which is consolidated during the compile into a separate file. This file has the same root name as the source file and is given a ".dbg" extension.

The Disassembler can now be executed using the target program (the compiled program).

Automatic Option Memory File

At the beginning of each disassembly session, the Disassembler searches for a file named "dis.cmd". If the file exists in the current directory, the Disassembler uses the contents of this file to set its options. If the file is not found, the default option profile is used. At the end of the disassembly session, all options in effect are written to the file.

Using the Disassembler

The Disassembler can be executed in these ways:

- from the command line with one or more options specified
- from the command line with only the default settings in effect
- from the menu
- from a command file containing Disassembler options.

From the Command Line — with Options

The Disassembler can be invoked from the command line with one or more options specified. The default settings, which may be changed, are:

- output displayed on screen
- no address listing
- no hex code listing
- no variable type listing
- no variable location listing.

The format for running the Disassembler from the command line with one or more options specified is:

disasm *+i=filename +m=module option [option] ...*

+i=filename

specifies an input file that contains the program or submodule to be disassembled.

+m=module

specifies the entry point to be disassembled. The entry point is searched for in the symbol table. If symbolic information is available for this entry point, the information is incorporated in the disassembly.

Note: The #, when used for the entry point, causes the entire program to be disassembled.

option

may be any of the following:

+a

ABSOLUTE ADDRESS LISTING

instructs the Disassembler to include an absolute address listing in the output. The default is no absolute address listing (-a).

+d=filename

SYMBOLIC DEBUGGER SYMBOLS

specifies a file of Symbolic Debugger symbols to be used in the disassembly. The input file name is used by default. The file name extension defaults to ".dbg".

+o=filename

OUTPUT FILE FOR DISASSEMBLY

specifies an output file to be used for disassembly. The input file name is used by default. The file name extension defaults to ".dis".

- +p=cmdfile** **OPTION FILE**
specifies a file from which the Disassembler can read its options. The default name is "dis.cmd".
- +r** **HEX CODE (RAW DATA) LISTING**
instructs the Disassembler to include a hex code (raw data) listing in the output. The default is no hex code listing (-r).
- s** **NO OUTPUT DISPLAY ON SCREEN**
instructs the Disassembler not to display the output on the screen. The default is output displayed on the screen (+s).
- +t** **VARIABLE TYPE LISTING**
instructs the Disassembler to include a variable type listing in the output. The default is no variable type listing (-t).
- +v** **VARIABLE LOCATION LISTING**
instructs the Disassembler to include a variable location listing in the output. The default is no variable location listing (-v).

Option	Function
+a	Absolute address listing
+d=filename	Symbolic debugger symbols
+o=filename	Output file for disassembly
+p=cmdfile	Option file
+r	Hex code (raw data) listing

Figure 6-1 (Part 1 of 2). Disassembler Command-Line Options

Option	Function
-s	No output display on screen
+t	Variable type listing
+v	Variable location listing

Figure 6-1 (Part 2 of 2). Disassembler Command-Line Options

The Disassembler command-line options, procedure names, and module names can be in uppercase or lowercase. However, case is significant in the specified file names. File names can be either uppercase or lowercase, but the case has to correspond to the case used in the operating system. For example:

```
disasm +a +i=INFILE +m=MOD +d=DBGFILE +o=OUTFILE.OUT
```

Note: The same command could be executed with a # substituted for the module name (+m = #), resulting in the entire program being disassembled.

Output files may use any extension. However, if an extension is not specified, the default extension ".dis" is used. An input file can be a compiled source program or input file that does not have an extension, or the debug file that has a ".dbg" extension.

If the root of the symbol file name and the input file name are the same, only the root of the name needs to be specified, as in the command:

```
disasm p1 SAMPLE
```

Example:

{For this example, PROGRAM SAMPLE is located in the file "p1.pas". After the program is compiled, the executable file is "p1"}

```

PROGRAM SAMPLE (INPUT,OUTPUT);
VAR X,Y,Z : INTEGER;
BEGIN
  X := 1;
  Y := X+1;
  WRITELN('X= ',X,' Y= ',Y);
END.

```

The following code creates the assembler equivalent of the SAMPLE program and writes the output to the file named "out.dis". The address, hex code, variable type, and variable location listings are omitted. Provided that the symbol file and the input file are created with the same name (p1), the command form is:

```
disasm +i=p1 +m=SAMPLE +o=out
```

By default, the following output is displayed on the screen:

```

                                XDEF      sample
                                XREF      .r_init
                                XREF      .r_wssf
                                XREF      .r_iochk
                                XREF      .r_wisf
                                XREF      .r_end
*
*

SAMPLE:      STM      R6,$FFB4(R1)
              CAL     R1,$FF60(R1)
              LR      R14,R0
              L       R13,$1(R14)
              LR      R4,R13
              AI      R5,R1,$54
              BALIX   R15,.r_init
              L       R0,$8(R14)

USERCODE:    LIS     R15,$1
              ST      R15,$BC0(R13)

LN_5:        AI      R15,R13,$BC0
              L       R3,$0(R15)
              AIS     R3,$1
              ST      R3,$0(R15)

```

```

LN_6:      AI      R2,R14,$C
           LR      R15,R13
           AI      R4,R1,$FFEC
           ST      R15,$0(R4)
           BALIX   R15,.r_wssf
           L       R0,$10(R14)
           LR      R15,R13
           AI      R5,R1,$FFEC
           ST      R15,$0(R5)
           BALIX   R15,.r_iochk
           L       R0,$14(R14)
           L       R2,$BC0(R13)
           LR      R5,R13
           AI      R4,R1,$FFEC
           ST      R5,$0(R4)
           BALIX   R15,.r_wisf
           L       R0,$18(R14)

EXIT:      BALIX   R15,.r_end
           L       R0,$28(R14)
           LM      R6,$54(R1)
           BRX     R15
           CAL     R1,$A0(R1)

```

Note: The output file "out.dis" contains:

```

1 ---> * Disassembled Instruction Code
        * Options in effect:
        *   Address listing?                [N]
        *   Hex code listing?               [N]
        *   Variable type listing?          [Y]
        *   Variable location listing?      [Y]

2 ---> * Input file: p1
        * Debug file: p1.dbg
        * Module:      sample

3 ---> * Initial address: $100002D0
        * Final   address: $10000356
        *

```



```

XDEF      sample
XREF      .r_init
XREF      .r_wssf
XREF      .r_iochk
XREF      .r_wisf
XREF      .r_end

*
*

SAMPLE:    STM      R6,$FFB4(R1)
           CAL      R1,$FF60(R1)
           LR       R14,R0
           L        R13,$1(R14)
           LR       R4,R13
           AI       R5,R1,$54
           BALIX    R15,.r_init
           L        R0,$8(R14)

USERCODE:  LIS      R15,$1
           ST       R15,$BC0(R13)

LN_5:      AI       R15,R13,$BC0
           L        R3,$0(R15)
           AIS      R3,$1
           ST       R3,$0(R15)

LN_6:      AI       R2,R14,$C
           LR       R15,R13
           AI       R4,R1,$FFEC
           ST       R15,$0(R4)
           BALIX    R15,.r_wssf
           L        R0,$10(R14)
           LR       R15,R13
           AI       R5,R1,$FFEC
           ST       R15,$0(R5)
           BALIX    R15,.r_iochk
           L        R0,$14(R14)
           L        R2,$BC0(R13)
           LR       R5,R13
           AI       R4,R1,$FFEC
           ST       R5,$0(R4)
           BALIX    R15,.r_wisf
           L        R0,$18(R14)

```

```

EXIT:      BALIX      R15, .r_end
           L          R0, $28(R14)
           LM         R6, $54(R1)
           BRX        R15
           CAL        R1, $A0(R1)

```

The numbered arrows correspond to the following:

1. Effective option listing.
2. Input file names. By default, the debug file and the input file have the same name.
3. Initial and final addresses.

From the Command Line — without Options

The Disassembler can be invoked without options. In this case, output is written to the screen and no address, hex code, variable type, or variable location listings are included.

The format for running the Disassembler from the command line without options is:

```
disasm input-file-name entry-point
```

input-file-name

is the file that contains the program or submodule to be disassembled.

entry-point

is the name of the procedure or function to be disassembled.

Both the *input-file-name* and the *entry-point* parameters are required; specifying only one is a syntax error.

The following example uses the SAMPLE program and the executable file "p1". To disassemble SAMPLE, enter:

```
disasm p1 SAMPLE
```

or

```
disasm p1 #
```

The following is displayed on the screen:

```

                                XDEF      sample
                                XREF      .r_init
                                XREF      .r_wssf
                                XREF      .r_iochk
                                XREF      .r_wisf
                                XREF      .r_end
*
*
1 --->  SAMPLE:      STM      R6,$FFB4(R1)
                                CAL      R1,$FF60(R1)
                                LR        R14,R0
                                L         R13,$1(R14)
                                LR        R4,R13
                                AI         R5,R1,$54
                                BALIX     R15,.r_init
                                L         R0,$8(R14)

2 --->  USERCODE:   LIS      R15,$1
                                ST        R15,$BC0(R13)

3 --->  LN_5:        AI         R15,R13,$BC0
                                L         R3,$0(R15)
                                AIS       R3,$1
                                ST        R3,$0(R15)

                                LN_6:      AI         R2,R14,$C
                                LR        R15,R13
                                AI         R4,R1,$FFEC
                                ST        R15,$0(R4)
```



```

4 --->          BALIX    R15,.r_wssf
                  L        R0,$10(R14)
                  LR       R15,R13
                  AI       R5,R1,$FFEC
                  ST       R15,$0(R5)
                  BALIX    R15,.r_iochk
                  L        R0,$14(R14)
                  L        R2,$BC0(R13)
                  LR       R5,R13
                  AI       R4,R1,$FFEC
                  ST       R5,$0(R4)
                  BALIX    R15,.r_wisf
                  L        R0,$18(R14)

5 --->  EXIT:      BALIX    R15,.r_end
                  L        R0,$28(R14)
                  LM       R6,$54(R1)
                  BRX      R15
                  CAL      R1,$A0(R1)

```

The numbered arrows correspond to the following:

1. The Disassembler uses a specified entry point to label the first line of the assembler code.
2. The label "USERCODE" marks the beginning of the program corresponding to "x := 1 ;".
3. The label "LN_n" marks the beginning of the *n*'th line of the high-level source code.
4. System routine ".r_wssf" is referenced.
5. The beginning of the exit code is labeled "EXIT".

From the Menu System

Options can be selected from a system of menus. To invoke the main menu, enter:

```
disasm
```

The following menu appears on the screen:

```
**** Main Menu ****
-----

1 ... Select input options
2 ... Produce disassembly
3 ... Select output form options
4 ... Select output designation
5 ... Display options in effect

Enter Selection # (or q to Quit): _
```

Any option may be selected from the menu. Press the Enter key after each selection. For example, to display the options currently in effect, select option 5 from the menu. This menu is illustrated in "Display Options Selection." The default profile shows the options initially in effect.

Input Options Selection

If option 1 is selected, the "Input Options Menu" appears:

```
**** Input Options Menu ****
```

```
-----
```

- 1 ... Specify input file name []
- 2 ... Specify program or entry point []
- 3 ... Specify debug symbol file []

```
Enter Selection # (or <Enter> for Main Menu): _
```

Empty brackets at the end of options 1, 2, and 3 indicate that the input options have not yet been selected.

The SAMPLE program resides in input file "p1". To specify input file "p1", enter option 1. The response is:

```
**** Input Options Menu ****
```

```
-----
```

- 1 ... Specify input file name [] .
- 2 ... Specify program or entry point []
- 3 ... Specify debug symbol file []

```
Enter Selection # (or <Enter> for Main Menu): 1
```

```
Enter input file name  
or <Enter> to cancel: p1
```

No extension is required for the input file; you need only to enter "p1".

The updated screen shows:

```
**** Input Options Menu ****
-----

1 ... Specify input file name      [p1]
2 ... Specify program or entry point [ ]
3 ... Specify debug symbol file    [p1.dbg]

Enter Selection # (or <Enter> for Main Menu): _
```

If SAMPLE is used as the entry point (option 2), the screen is updated once again to include the entry point information. When option 2 is selected, this screen is displayed:

```
**** Input Options Menu ****
-----

1 ... Specify input file name      [p1]
2 ... Specify program or entry point [ ]
3 ... Specify debug symbol file    [p1.dbg]

Enter Selection # (or <Enter> for Main Menu): 2

Enter target name
(program, or submodule name)
or <Enter> to cancel:                SAMPLE
```

A # may be used as the entry point for option 2, in which case the entire program is disassembled.

After the input file name (p1) and the program name (SAMPLE) are entered, it is possible to return to the main menu to disassemble the program, using only the Disassembler's default settings, by choosing the "Produce disassembly" selection.

Produce Disassembly Selection

After the selections for the "Input Options Menu" are complete, the Disassembler can be executed using its default settings. It is not necessary to continue through the menus unless changes are to be made to the default settings or to the output designation.

When the "Produce disassembly" option is selected, the screen is cleared and the disassembled output is displayed. Upon completion, this message is displayed:

Do you wish to continue? (y/n)

If a "y" is entered, the main menu is displayed once again. You can now change the default settings for the output form by selecting option 3, or change the output designation by selecting option 4. By selecting option 1, another program can be disassembled.

If an "n" is entered, the Disassembler program is terminated.

Output Form Options Selection

If option 3 is selected, the "Output Form Options" menu appears:

```
**** Output Form Options ****
```

```
-----
```

- | | |
|---------------------------------|-----|
| 1 ... Address listing | [n] |
| 2 ... Hex code listing | [n] |
| 3 ... Variable type listing | [n] |
| 4 ... Variable location listing | [n] |

Enter Selection # (or <Enter> for Main Menu): _

To change any of the default settings, enter the appropriate option number. For example, to include an address listing, select option 1. The response is:

```
**** Output Form Options ****
```

```
-----  
1 ... Address listing [n]  
2 ... Hex code listing [n]  
3 ... Variable type listing [n]  
4 ... Variable location listing [n]
```

```
Enter Selection # (or <Enter> for Main Menu): 1
```

```
Include address listing? (y/n)
```

Enter "y" to include an address listing. The updated screen shows that the change has been made.

Output Designation Selection

If option 4 is selected, the "Output Designation Menu" appears:

```
**** Output Designation Menu ****
```

```
-----  
1 ... Write output to file [ ]  
2 ... Display output on screen [y]
```

```
Enter Selection # (or <Enter> for Main Menu): _
```


To write to a file, select option 1. The following screen is displayed:

```
**** Output Designation Menu ****
-----
1 ... Write output to file          [ ]
2 ... Display output on screen      [y]

Enter Selection # (or <Enter> for Main Menu): 1

Enter output file name
or <Enter> to cancel:
```

It is possible to select both options in this menu; the output is then written to a file and displayed on the screen.

Display Options Selection

If option 5 is selected, the "Options in effect" screen is displayed. This screen contains a list of the options that are currently being used by the Disassembler. These options are selected from the "Output Form Options" menu and the "Output Designation Menu".

```
Options in effect:
- Disassemble high level program or
  submodule
  Input file:  p1
  Module:      SAMPLE
  Debug File:  p1.dbg
- Include variable type definitions
- Include variable location listing
- Display output on screen

Press Enter to Continue ...
```

From a Command File

A command file can be created that contains options readable by the Disassembler. The file is created with an editor, and options are entered one option per line.

The following command file example produces pure assembler code with the variable type and location information in comment form in the output file. This command file contains:

```
+i=p1
+m=SAMPLE
+o=out
-s
+t
+v
```

Note: In this example, "+m=SAMPLE" may be replaced with "+m= #" to disassemble the entire program.

If this command file is named "COMMAND.CMD", the Disassembler command is:

```
disasm +p=COMMAND.CMD
```

The output file "out" uses the default extension ".dis". The output file contains:

```
* Disassembled Instruction Code
* Options in effect:
*   Address listing?                [n]
*   Hex code listing?              [n]
*   Variable type listing?         [y]
*   Variable location listing?     [y]

* Input file: p1
* Debug file: p1.dbg
* Module:      SAMPLE

* Initial address: $100002D0
* Final   address: $10000356
*
```

```

XDEF      sample
XREF      .r_init
XREF      .r_wssf
XREF      .r_iochk
XREF      .r_wisf
XREF      .r_end
*
*
SAMPLE:    STM      R6,$FFB4(R1)
           CAL      R1,$FF60(R1)
           LR       R14,R0
           L        R13,$1(R14)
           LR       R4,R13
           AI       R5,R1,$54
           BALIX    R15,.r_init
           L        R0,$8(R14)

USERCODE:  LIS      R15,$1
           ST       R15,$BC0(R13)

LN_5:      AI       R15,R13,$BC0
           L        R3,$0(R15)
           AIS      R3,$1
           ST       R3,$0(R15)

LN_6:      AI       R2,R14,$C
           LR       R15,R13
           AI       R4,R1,$FFEC
           ST       R15,$0(R4)
           BALIX    R15,.r_wssf
           L        R0,$10(R14)
           LR       R15,R13
           AI       R5,R1,$FFEC
           ST       R15,$0(R5)
           BALIX    R15,.r_iochk
           L        R0,$14(R14)
           L        R2,$BC0(R13)
           LR       R5,R13
           AI       R4,R1,$FFEC
           ST       R5,$0(R4)
           BALIX    R15,.r_wisf
           L        R0,$18(R14)

```



```

EXIT:      BALIX      R15,.r_end
           L          R0,$28(R14)
           LM         R6,$54(R1)
           BRX        R15
           CAL        R1,$A0(R1)

*
* offset definitions
*
* entry SAMPLE user name SAMPLE
* entry code begins at $100002D0
* user code begins at $100002E8
* exit code begins at $10000340
* addresses for source code by line number:
*   5: $100002EE      6: $100002F8
*
*
* variable definitions:
*
* sample:
*   X                type -3 $00000BC0 in //global
*   Y                type -3 $00000BBC in //global
*   Z                type -3 $00000BB8 in //global
*
*
* type definitions:
*
*
*   -1 = integer (1 byte)
*   -2 = integer (2 bytes)
*   -3 = integer (4 bytes)
*   -4 = unsigned integer (1 byte)
*   -5 = unsigned integer (2 bytes)
*   -6 = unsigned integer (4 bytes)
*   -7 = character (1 byte)
*   -8 = character (2 bytes)
*   -9 = single precision floating point (4 bytes)
*  -10 = double precision floating point ( 8 bytes)
*  -11 = logical (1 byte)
*  -12 = logical (2 bytes)
*  -13 = logical (4 bytes)
*  -14 = file
*  -15 = complex floating point (16 bytes)
*  -16 = double complex floating point (32 bytes)
00000000 .... ....                                END

```

Appendix A. Messages

Pascal contains a file of compile-time error messages named `/usr/include/vspctmsg.inc`. The compiler generates error numbers and messages if this file is present in the default directory and errors are encountered.

If the *efilename* command-line option is used, error messages are written to the error file. Otherwise, error messages are displayed on the console.

Compile-Time Lexical Messages

- | | |
|----|---|
| 1 | Unknown environment |
| 2 | Bad option |
| 3 | Bad margins or sequence field |
| 4 | Can't open listing file |
| 5 | Can't open source file |
| 6 | No source file |
| 7 | Can't open i-code file |
| 8 | Can't open error file |
| 9 | Can't call |
| 10 | Too many digits |
| 11 | Digit expected after '.' in a real number |
| 12 | Integer overflow |
| 13 | Digit expected in the exponent of a real number |
| 14 | End-of-line encountered in a string constant |
| 15 | Invalid character in input |
| 16 | Premature end-of-file in source program |
| 17 | Extra characters encountered after the end of the program |
| 18 | End-of-file encountered in a comment |

Compile-Time Syntactic Messages

20	Invalid symbol
21	Error in simple type
22	Error in declaration part
23	Error in parameter list of a procedure or function
24	Error in constant
25	Error in type
26	Error in field list of a record declaration
27	Error in factor of an expression
28	Error in variable
29	Identifier expected
30	Integer expected
31	'(' expected
32	')' expected
33	'[' expected
34	']' expected
35	':' expected
36	';' expected
37	'=' expected
38	',' expected
39	'*' expected
40	':=' expected
41	program keyword expected
42	of keyword expected
43	begin keyword expected
44	end keyword expected
45	then keyword expected
46	until keyword expected
47	do keyword expected
48	to or downto keyword expected
50	if keyword expected
51	. (period) expected
52	implementation keyword expected
53	interface keyword expected

Compile-Time Semantic Messages

- 100 Identifier declared twice in the same block
- 101 Identifier is not of the appropriate class
- 102 Identifier not declared
- 103 Sign not allowed
- 104 Number expected
- 105 Lower bound exceeds upper bound
- 106 Incompatible subrange types
- 107 Type of constant must be integer
- 108 Type must not be real
- 109 Tagfield must be a scalar or subrange

- 110 Type incompatible with tagfield type
- 111 Index type must not be real
- 112 Index type must be scalar or subrange
- 113 Index type must not be integer or longint
- 114 Unsatisfied forward reference
- 115 Forward reference type identifier cannot appear in a variable declaration
- 116 Forward declaration -- repetition of parameter list not allowed
- 117 Forward declaration function -- repetition of result type not allowed
- 118 Function result type must be scalar, subrange or pointer
- 119 File is not allowed as a value parameter

- 120 Missing result type in function declaration
- 121 F-format for real type only
- 122 Error in type of parameter to a standard function
- 123 Error in type of parameter to a standard procedure
- 124 Number of actual parameters does not agree with declaration
- 125 Invalid parameter substitution
- 126 Result type of parametric function does not agree with declaration
- 127 Expression is not of set type
- 128 Only tests for equality allowed
- 129 Strict inclusion not allowed

- 130 Comparison of file variables not allowed
- 131 Invalid type of operand(s)
- 132 Operand type must be boolean

- 133 Set element type must be scalar or subrange
- 134 Set element types not compatible
- 135 Type of variable is not array or string
- 136 Index type is not compatible with declaration
- 137 Type of variable is not record
- 138 Type of variable must be file or pointer
- 139 Invalid type of loop control variable

- 140 Invalid expression type
- 141 Assignment of files not allowed
- 142 Case selector incompatible with selecting expression
- 143 Subrange bounds must be scalar
- 144 Operand type conflict
- 145 Assignment to standard function is not allowed
- 146 Assignment to formal function is not allowed
- 147 No such field in this record
- 148 Type error in read
- 149 Actual parameter must be a variable

- 150 Multiply defined case selector
- 151 Missing corresponding variant declaration
- 152 Real or string tagfields not allowed in variant record
- 153 Previous declaration was not forward
- 154 Substitution of standard procedure or function not allowed
- 155 Multiple defined label
- 156 Multiple declared label
- 157 Undefined label
- 158 Undeclared label
- 159 Value parameter expected

- 160 Multiple defined record variant
- 161 File not allowed here
- 162 Unknown compiler directive (not external or forward)
- 163 Variable cannot be a packed field
- 164 Set of real is not allowed
- 165 A field of a packed record cannot be a var parameter
- 166 Case selector expression must be a scalar or a subrange
- 167 String sizes must be equal
- 168 String too long
- 169 Value out of range

170 Cannot take the address of a standard procedure or function
171 Assignment to function result must be done inside that function
172 Control variable of a for statement must be local
175 File variable expected
176 Must be within the procedure or function that is being EXITed
177 Cannot pass cexternal as procedure or function parameter

180 Short reals are not allowed in CONSTANT or VALUE
181 Repetition factors are not allowed in record fields
182 Repetition factors must be positive integer constant expressions
183 Incompatible type in CONSTANT or VALUE
184 Too many fields defined in structured constants
185 Only STATIC and DEF/REF variables may be initialized with
VALUE

190 No such unit in this file
191 File variable must be of type text
192 Type error in READSTR
193 Type error in WRITESTR
194 F-Format not valid for specified variable type
195 Attempt to pad beyond the defined string length
196 READSTR must have at least one variable after the source string
197 Variable after comma symbol not found
198 WRITESTR must have at least one variable after the source string

200 Invalid standard procedure or function in current mode
201 Packed array too long for conversion to string
203 Assignment to constants not allowed
205 Index will make resulting array of PACK/UNPACK too big
207 Incompatible index parameter passed to PACK/UNPACK

211 Error in conversion of type to constant

217 Array parameters passed to PACK/UNPACK not of the same type

221 Invalid function call in constant expression
222 Error in parameter to constant function

230 Division by zero

250 Invalid syntax for compiler directive

- 251 Line length exceeds maximum
- 276 Conformant string must be a VAR or CONST parameter

Compiler Limitation Messages

- 300 Too many nested record scopes
- 301 Set limits out of range
- 302 String limits out of range
- 303 Too many nested procedures or functions
- 304 Too many nested include or uses files
- 305 Include not allowed in interface section
- 306 Pack and unpack are not implemented
- 307 Too many units
- 308 Set constant out of range
- 309 Maximum comparable packed array of char type is of size 255

- 310 Too many nested with statements
- 311 Too many nested function calls
- 312 Record too big(maximum size is 32766 bytes)
- 313 Too many elements in an array(maximum size or elements is 32766)
- 314 Too many variables in one scope (maximum is 32766)

- 350 Procedure too large
- 351 File name in option too long
- 352 Conflicting options : optimization has been disabled

Input/Output Messages

- 400 Not enough room for code file
- 401 Error in rereading code file
- 402 Error in reopening text file
- 403 Unable to open uses file
- 404 Error in reading uses file
- 405 Error in opening include file
- 406 Error in rereading previously read text block
- 407 Not enough room for intermediate code file
- 408 Error in writing code file
- 409 Error in reading intermediate code file
- 410 Unable to open listing file

- 900 OPTIMIZER ERROR PHASE 0 degrade optimization level
- 901 OPTIMIZER ERROR PHASE 1 degrade optimization level
- 902 OPTIMIZER ERROR PHASE 2 degrade optimization level
- 903 OPTIMIZER ERROR PHASE 3 degrade optimization level
- 904 OPTIMIZER ERROR PHASE 4 degrade optimization level
- 905 OPTIMIZER ERROR PHASE 5 degrade optimization level
- 906 OPTIMIZER ERROR PHASE 6 degrade optimization level
- 907 OPTIMIZER ERROR PHASE 7 degrade optimization level
- 908 OPTIMIZER ERROR PHASE 8 degrade optimization level
- 909 OPTIMIZER ERROR PHASE 9 degrade optimization level

- 1000 Could not do block write on outfile
- 1001 Could not do block read on outfile
- 1002 Could not do block read on infile
- 1003 Could not seek to block requested in infile
- 1004 No more memory
- 1005 Code not implemented yet
- 1006 FATAL CODE GENERATOR ERROR
- 1007 Unable to open input file *.i
- 1008 Unable to open output file *.obj
- 1009 Input file is not a .i file
- 1010 Input file is not a correct version

Appendix B. ASCII Character Set

This appendix lists the standard ASCII characters in numerical order with the corresponding decimal, octal, and hexadecimal values. The control characters are indicated by a "Ctrl-" notation. For example, the horizontal tab (HT) is indicated by "Ctrl-I", which is keyed by simultaneously pressing the Ctrl key and I key.

Note that this character set was originally developed for teletype communications. Consequently, most of the original control characters (decimal 0 through 31) are undefined in other types of communication. However, two important control characters have retained their original function: LF (decimal 10), which generates a line feed (causing subsequent output on a display or printer to appear on the next line), and CR (decimal 13), which generates a carriage return.

Decimal Value	Octal Value	Hex Value	Control Character	ASCII Symbol	Meaning
0	000	00	Ctrl-@	NUL	null
1	001	01	Ctrl-A	SOH	start of heading
2	002	02	Ctrl-B	STX	start of text
3	003	03	Ctrl-C	ETX	end of text
4	004	04	Ctrl-D	EOT	end of transmission
5	005	05	Ctrl-E	ENQ	inquiry
6	006	06	Ctrl-F	ACK	acknowledge
7	007	07	Ctrl-G	BEL	bell

Figure B-1 (Part 1 of 6). ASCII Character Set

Decimal Value	Octal Value	Hex Value	Control Character	ASCII Symbol	Meaning
8	010	08	Ctrl-H	BS	backspace
9	011	09	Ctrl-I	HT	horizontal tab
10	012	0A	Ctrl-J	LF	line feed
11	013	0B	Ctrl-K	VT	vertical tab
12	014	0C	Ctrl-L	FF	form feed
13	015	0D	Ctrl-M	CR	carriage return
14	016	0E	Ctrl-N	S0	shift out
15	017	0F	Ctrl-O	SI	shift in
16	020	10	Ctrl-P	DLE	data link escape
17	021	11	Ctrl-Q	DC1	device control 1
18	022	12	Ctrl-R	DC2	device control 2
19	023	13	Ctrl-S	DC3	device control 3
20	024	14	Ctrl-T	DC4	device control 4
21	025	15	Ctrl-U	NAK	negative acknowledge
22	026	16	Ctrl-V	SYN	synchronous idle
23	027	17	Ctrl-W	ETB	end of transmission block
24	030	18	Ctrl-X	CAN	cancel
25	031	19	Ctrl-Y	EM	end of medium
26	032	1A	Ctrl-Z	SUB	substitute
27	033	1B	Ctrl-[ESC	escape
28	034	1C	Ctrl-\	FS	file separator
29	035	1D	Ctrl-]	GS	group separator
30	036	1E	Ctrl-^	RS	record separator
31	037	1F	Ctrl- _~	US	unit separator
32	040	20		SP	space
33	041	21		!	
34	042	22		"	
35	043	23		#	

Figure B-1 (Part 2 of 6). ASCII Character Set

Decimal Value	Octal Value	Hex Value	Control Character	ASCII Symbol	Meaning
36	044	24		\$	
37	045	25		%	
38	046	26		&	
39	047	27		'	apostrophe
40	050	28		(
41	051	29)	
42	052	2A		*	
43	053	2B		+	
44	054	2C		,	comma
45	055	2D		-	minus
46	056	2E		.	period
47	057	2F		/	
48	060	30		0	
49	061	31		1	
50	062	32		2	
51	063	33		3	
52	064	34		4	
53	065	35		5	
54	066	36		6	
55	067	37		7	
56	070	38		8	
57	071	39		9	
58	072	3A		:	
59	073	3B		;	
60	074	3C		<	
61	075	3D		=	
62	076	3E		>	
63	077	3F		?	

Figure B-1 (Part 3 of 6). ASCII Character Set

Decimal Value	Octal Value	Hex Value	Control Character	ASCII Symbol	Meaning
64	100	40		@	
65	101	41		A	
66	102	42		B	
67	103	43		C	
68	104	44		D	
69	105	45		E	
70	106	46		F	
71	107	47		G	
72	110	48		H	
73	111	49		I	
74	112	4A		J	
75	113	4B		K	
76	114	4C		L	
77	115	4D		M	
78	116	4E		N	
79	117	4F		O	
80	120	50		P	
81	121	51		Q	
82	122	52		R	
83	123	53		S	
84	124	54		T	
85	125	55		U	
86	126	56		V	
87	127	57		W	
88	130	58		X	
89	131	59		Y	
90	132	5A		Z	
91	133	5B		[

Figure B-1 (Part 4 of 6). ASCII Character Set

Decimal Value	Octal Value	Hex Value	Control Character	ASCII Symbol	Meaning
92	134	5C		\	
93	135	5D]	
94	136	5E		^	
95	137	5F		_	underscore
96	140	60		`	grave
97	141	61		a	
98	142	62		b	
99	143	63		c	
100	144	64		d	
101	145	65		e	
102	146	66		f	
103	147	67		g	
104	150	68		h	
105	151	69		i	
106	152	6A		j	
107	153	6B		k	
108	154	6C		l	
109	155	6D		m	
110	156	6E		n	
111	157	6F		o	
112	160	70		p	
113	161	71		q	
114	162	72		r	
115	163	73		s	
116	164	74		t	
117	165	75		u	
118	166	76		v	
119	167	77		w	

Figure B-1 (Part 5 of 6). ASCII Character Set

Decimal Value	Octal Value	Hex Value	Control Character	ASCII Symbol	Meaning
120	170	78		x	
121	171	79		y	
122	172	7A		z	
123	173	7B		{	
124	174	7C			
125	175	7D		}	
126	176	7E		~	
127	177	7F		DEL	delete

Figure B-1 (Part 6 of 6). ASCII Character Set

Appendix C. Migrating Programs

IBM RT PC VS Pascal is source-language compatible with IBM Pascal/VS Release 2.2. All Pascal/VS Release 2.2 statements, data types, and compiler directives except partitioned data sets are supported. Most Pascal/VS Release 2.2 programs may be recompiled on the RT PC and executed without modification. Some Pascal/VS Release 2.2 compiler directives are accepted syntactically but not functionally.

Most Pascal programs written in Pascal/VS Release 2.2 do compile and execute on the RT PC VS Pascal compiler; however, because of hardware architecture differences, operating system differences, and compiler implementation differences, some of these programs may produce unintended results.

The purpose of this chapter is to describe those areas of the compiler where differences occur so that you can determine the extent to which programs might be affected, and understand what changes are required to achieve the intended results.

Differences between Pascal/VS Release 2.2 and RT PC VS Pascal may cause unintended results in the following areas.

Floating-Point Representation

Precision of Results: The IBM System 370 floating-point representation is different from that of the RT PC. The RT PC uses IEEE standard floating-point representation and calculations. The IBM System 370 representation is different from the IEEE representation for floating point in the number of bits used to represent the mantissa and exponent of a number; therefore the precision with which the number is stored is different. In addition, there may be differences in the algorithms used to compute mathematical functions which could lead to different results near the limits of precision.

Exception Handling: The IEEE standard calls for floating-point exceptions (overflow, underflow, undefined) to be reported by returning a particular bit pattern as a result (+infinity, -infinity, Not-a-Number) rather than raising an actual exception condition.

Representation Dependence: Programs that map floating point variables onto other data types and depend on the bitwise floating point representation may produce unintended results.

Output Format: The ANSI Pascal standard does not precisely specify the output format for floating point numbers in all details. As a result, the output format may differ in some instances.

Character Representation

Environments: Pascal/VS Release 2.2 operates in EBCDIC environments, and RT PC VS Pascal operates in ASCII.

Character Data Values: Programs that depend on a particular data value for a character, or a particular relationship among character data values, may produce unintended results.

Collating Sequence: Programs that depend on the order of character values to sort or otherwise work with character data may produce unintended results.

Binary Files with Embedded Character Data: When porting a Pascal created binary data file from an IBM System 370 to an RT PC, the data file may mix character and numeric data; the characters remain in EBCDIC.

Run-time Messages

Error Numbers, Contexts, Message Texts: Error numbers, contexts, and message texts are different in IBM RT PC VS Pascal and IBM Pascal/VS.

Data Storage

Uninitialized Data: Programs that depend on the value of uninitialized storage may produce unintended results. Previously used storage uninitialized in a particular subroutine may also produce unintended results.

Logical Representation: Programs that depend on the internal representation of LOGICAL data values may produce unintended results.

Storage Mapping: Programs that deliberately ignore the boundary of a data element and index out of one array and into another may produce unintended results. In general, programs that depend on the storage layout or alignment of data may produce unintended results.

Files

Binary Files Not Pure: When porting a Pascal-created binary data file from an IBM System 370 to an RT PC, the internal format of the data file may be different. Data files containing characters or floating point numbers must be mapped by a translate utility if they are to be ported.

File Names: Case is significant in RT PC AIX, but not significant in the IBM System 370 environments.

Function Calls

Function Results When No Assignment Is Made: Programs that depend on a particular function result value, (such as 0) when no assignment to the function has been made may produce unintended results.

Order of Evaluation of Parameter Expressions: Programs that depend on parameter expressions being evaluated in a particular order (because of side effects) may produce unintended results.

Index

Special Characters

\$%+ compiler directive 2-13

A

+a Disassembler option 6-3
absolute address listing 6-3
address listing 6-3
AIX operating system 1-2
 programs under AIX 1-2
ANSI mode 1-2, 2-4
 command-line option 2-4
array representation 4-7
array storage 5-3
ASCII character set B-1
assembly language 6-1
automatic option memory file 6-2

B

b command-line option 2-3
binary code 2-1
binary files
 See files
binary files with embedded character data C-2
boolean variable 4-1
Bourne shell 3-2

C

C calling FORTRAN and Pascal 5-22
c command-line option 2-3
\$C- compiler directive 2-9
C shell 3-2
calling a routine 5-13
case significance 6-5
case, using 3-3
character data values C-2
character representation C-2
character set, ASCII B-1
%check off compiler directive 2-9
%check on compiler directive 2-10
code generation off 2-9
code modules in segment 2-13
collating sequence C-2
column major order 5-3
command file, Disassembler 6-18
command-line options 2-3-2-8
 Disassembler 6-3-6-5
comment toggles 2-8
compile information 2-13
compile information, generate 2-5
compiler 2-1, 2-8
 command-line options 2-3-2-6
 invoking 2-1
compiler directives 2-8-2-15
constant pool pointer 5-5, 5-13
%cpage compiler directive 2-10
Ctrl-D 3-10

D

- d+ command-line option 2-3, 6-1
- \$D+ compiler directive 2-10
- \$D- compiler directive 2-10
- +d Disassembler option 6-3
- data representations 4-1-4-8
- data storage C-3
- data types 5-1
- .dbg file 6-1, 6-3, 6-5
- DDNAME, open option 3-8
- debug 2-11
- debugger 2-4
- denormalized numbers 4-5
- .dis file 6-3, 6-5, 6-18
- dis.cmd file 6-2, 6-4
- Disassembler 2-3, 6-1-6-20
 - command file 6-18
 - command-line options 6-3-6-5
 - compiler directive 2-10
 - executing 6-2, 6-15
 - menus 6-12
 - preparation 6-1
 - symbolic disassembler information 2-3
 - with options specified 6-2
 - without options specified 6-9
- DOS shell 3-2

E

- \$E filename compiler directive 2-10
 - error file 2-10
- efilename command-line option 2-3
- entry code 5-12
- entry point 6-3
- environment variables 3-2
- environment-determined file names 3-1

- determined file names 3-2
 - using shell scripts 3-6
- environment-determined files 3-2
- environments C-2
- eof function 3-10
- error file 2-3
- error messages A-1-A-7
 - compile-time lexical A-1
 - compile-time semantic A-3
 - compile-time syntactic A-2
 - compiler limitation A-6
 - input/output A-7
- examples of programs
 - C calling FORTRAN and Pascal 5-22
 - FORTRAN calling Pascal and C 5-18
 - Pascal calling FORTRAN and C 5-14
- exception handling C-2
- executing a program 1-2-1-3
 - compilation process 1-4
- exit code 5-12
- extensions 6-5
- extreme values 4-5

F

- f+ command-line option 2-3
- &F+ compiler directive 2-10
- fields, sequence 2-5
- file names C-3
 - environment determined 3-1
 - program determined 3-1
- files
 - binary files not pure C-3
 - names of C-3
- floating-point expressions, checked 2-12
- floating-point hardware 2-4, 2-10
- floating-point registers 5-10
- floating-point representation

- exception handling C-2
- output format C-2
- precision of results C-1
- representation dependence C-2
- format
 - general-purpose output file 5-5
 - GPOFF 5-5
- FORTTRAN calling Pascal and C 5-18
- frame pointer 5-10
- function calls
 - parameter expression evaluation C-4
 - results when no assignment is made C-4
- function values 5-12

G

- g+ command-line option 2-4
- \$G+ compiler directive 2-10
- \$G- compiler directive 2-11
- general-purpose output file format 5-5
- general-purpose registers 5-10
- generate compile information 2-5
- GPOFF format 5-5

H

- hex code listing 6-4
- hexadecimal representation
 - selected numbers 4-6
- hidden bit 4-5

I

- \$I- compiler directive 2-11
- \$Ifilename compiler directive 2-11
- +i Disassembler option 6-3
- IBM mode 1-2, 2-4
 - command-line option 2-4
- identifiers, percent sign in 2-13
- %include compiler directive 2-11
- include source code 2-11
- include specified file 2-11
- +infinity 2-12, 4-2, 4-6, C-1
- infinity 4-2, 4-6, C-1
- input and output
 - See opening files
- input file 6-3, 6-5
- input options menu, Disassembler 6-12
- input parameter words 5-9
- input/output checks 2-11
- input/output primitives 5-4
- integer 4-2
- integer data representation 4-3
- INTERACTIVE, open option 3-9

L

- l+ command-line option 2-4
- \$L+ compiler directive 2-11
- \$L- compiler directive 2-11
- \$Lfilename compiler directive 2-11
- ld linker 2-1
- lfilename command-line option 2-4
- library
 - libvsfor.a 5-4
 - libvssys.a 5-4
- libvsfor.a 5-4
- libvssys.a 5-4

- lines per page 2-3
- lines, skipping 2-13
- linkage area 5-9
- linkage convention
 - See subroutine linkage convention
- listing file 2-1, 2-4
 - absolute address 6-3
 - file 2-4, 2-12
 - hex code 6-4
 - listing off, compiler directive 2-11
 - listing on, compiler directive 2-11
 - raw data 6-4
 - standard output device 2-4
 - titles in 2-13
 - variable location 6-4
 - variable type 6-4
- load module format 5-5
- local stack area 5-10
- location listing 6-4
- logical representation C-3

M

- +m Disassembler option 6-3
- machine-dependent optimization 2-7, 2-8
- machine-independent optimization 2-7, 2-8
- main menu, Disassembler 6-12
- man command-line option 2-4
- margins 2-3, 2-12
- %margins compiler directive 2-12
- matrix storage 5-3
- menu system, Disassembler 6-12
 - input options 6-12
 - options in effect 6-17
 - output designation 6-16
 - output form options 6-15
 - produce disassembly 6-15
- message texts C-3

- migrating programs C-1, C-4
- modes

- ANSI 1-2
- IBM 1-2

N

- \$N+ compiler directive 2-12
- NAME, open option 3-9
- not-a-number 2-12, 4-2, 4-6, C-1

O

- +o Disassembler option 6-3
- open options
 - DDNAME 3-8
 - INTERACTIVE 3-9
 - NAME 3-9
 - UCASE 3-9
- opening files
 - environment variables 3-2
 - program determined files 3-7
 - shell scripts 3-6
- optimization levels 2-4
- optimization of programs 2-7
- option file, Disassembler 6-4
- options in effect menu, Disassembler 6-17
- output designation menu 6-16
- output display, Disassembler 6-4
- output file format, general-purpose 5-5
- output file, disassembly 6-3
- output form options menu, Disassembler 6-15
- output format C-2
- output parameter words 5-10
- o1+ command-line option 2-4, 2-7
- o2+ command-line option 2-4, 2-7

o3+ command-line option 2-4, 2-7
o4+ command-line option 2-5, 2-7

P

+p Disassembler option 6-3
packing methods 4-8
%page compiler directive 2-12
page eject 2-10, 2-12
parameter addressing 5-12
parameter expressions, order of
evaluation C-4
parameter passing 5-11
.pas files 1-2
pascal
 See compiler
Pascal calling FORTRAN and C 5-14
pascterr file A-1
percent sign in identifiers 2-13
pointer representation 4-8
precision of results C-1
primitive input/output routines 5-4
%print off compiler directive 2-12
%print on compiler directive 2-12
produce disassembly selection,
 Disassembler 6-15
profiling 2-3
program determined file names 3-1
 determined files 3-7
program examples
 C calling FORTRAN and Pascal 5-22
 FORTRAN calling Pascal and C 5-18
 Pascal calling FORTRAN and C 5-14
program optimization 2-7
programs, migrating C-1

Q

\$Q- compiler directive 2-12

R

\$R+ compiler directive 2-13
+r Disassembler option 6-4
range checking
 off 2-10
 on 2-10, 2-13
raw data listing 6-4
real 4-2
real data representation 4-4
register save area 5-10
register usage 5-5
representation dependence C-2
reset procedure 3-10
routine calling 5-13
row major order 5-3
run-time library
 floating-point 2-4
run-time messages C-3

S

s command-line option 2-5
s- command-line option 2-5
-s Disassembler option 6-4
\$\$ segment compiler directive 2-13
scalar type 4-1
sdb, symbolic debugger 2-4
search for units 2-13
segment, placing code modules in 2-13
sequence fields 2-5

- no fields 2-5
- set representation 4-6
- shell scripts 3-6
 - using different files 3-7
 - using the same file name 3-6
- shortreal 4-2
- shortreal data representation 4-3
- %skip compiler directive 2-13
- skip lines 2-13
- source printed
 - off 2-12
 - on 2-12
- sourcefl 2-1
- stack frame 5-8
 - frame pointer 5-10
 - input parameter words 5-9
 - linkage area 5-9
 - local area 5-10
 - output parameter words 5-10
 - register save area 5-10
 - temporary area 5-10
 - total frame 5-11
- standard output device list 2-4
- storage allocation 4-1
- storage mapping C-3
- storage of arrays 5-3
- storage of matrices 5-3
- string representation 4-8
- subrange elements 4-1
- subroutine linkage convention 5-5-5-13
 - entry code 5-12
 - exit code 5-12
 - function values 5-12
 - load module format 5-5
 - parameter addressing 5-12
 - parameter passing 5-11
 - register usage 5-5
 - routine calling 5-13
 - stack frame 5-8
 - traceback 5-12
- suppress warning messages 2-5

- Symbolic Debugger 2-4, 5-12, 6-3
- syntax conventions 1-6

T

- +t Disassembler option 6-4
- temporary stack area 5-10
- termin procedure 3-10, 3-11
- terminal input/output 3-10
 - reset 3-10
 - termin 3-11
 - termout 3-11
- termout procedure 3-10, 3-11
- %title compiler directive 2-13
- title in listing 2-13
- toggles 2-8
- traceback 5-12
- type listing 6-4

U

- \$U filename compiler directive 2-13
- UCASE, open option 3-9
- uninitialized data C-3
- units, search for 2-13
- unpacked char 4-2

V

- v+ command-line option 2-5
- +v Disassembler option 6-4
- variable location listing 6-4
- variable type listing 6-4
- vsp shell script 2-2

W

w- command-line option 2-5
warning messages off 2-5
word values 4-1





IBM RT PC

Reader's Comment Form

IBM RT PC VS Pascal
User's Guide

SH23-0127-0

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

For prompt resolution to questions regarding setup, operation, program support, and new program literature, contact the authorized IBM RT PC dealer in your area.

Comments:

Tape

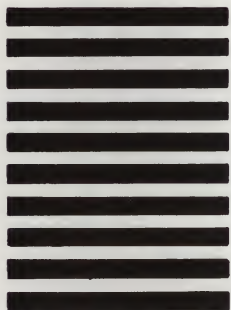
Please Do Not Staple

Tape

Cut or Fold Along Line

Fold and tape

Fold and tape



International Business Machines Corporation
Department 79L, Building 4
Commerce Park & Eagle Road
Danbury, Connecticut 06810

POSTAGE WILL BE PAID BY ADDRESSEE:

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

BUSINESS REPLY MAIL

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES





© IBM Corp. 1987
All rights reserved.

International Business Machines Corporation
Department 79L, Building 4
Commerce Park and Eagle Road
Danbury, CT 06810

Printed in the
United States of America

SH23-0127



SH23-0127-00



IBM RT PC

VS Pascal Reference Manual

Programming Family



Personal
Computer
Software

SH23-0128

IBM RT PC

VS Pascal Reference Manual

Programming Family



Personal
Computer
Software

First Edition (March 1987)

The information in this manual applies to Version 1 of IBM RT PC VS Pascal for use with Release 2.1 of the AIX Operating System, and it applies to all subsequent releases and modifications until otherwise indicated in new editions or Technical Newsletters.

Changes are made periodically to the information herein; these changes will be incorporated in new editions of this publication.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM program product in this publication is not intended to state or imply that only IBM's program product may be used. Any functionally equivalent program may be used instead.

International Business Machines Corporation provides this manual "as is" without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this manual at any time.

Requests for copies of this product and for technical information about the system should be made to your authorized IBM RT PC dealer.

A reader's comment form is provided at the back of this publication. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

© IBM Corporation 1987

™ RT PC is a trademark of IBM Corporation

™ AIX is a trademark of IBM Corporation

Preface

This reference manual contains a formal description of the Pascal programming language as implemented on the IBM RT PC¹ using the AIX² Operating System. ANSI standard language information appears in black print. Enhancements to the language are also described and are indicated by blue print.

Contents:

Chapter 1 — "Introduction" gives a general overview of Pascal and introduces the terms and concepts of the language.

Chapter 2 — "Data Types" describes the Pascal predefined standard types, user-defined types, structured types, and pointer types.

Chapter 3 — "Variables" describes the procedures for declaring, accessing, and referencing variables of different types.

Chapter 4 — "Expressions" describes the syntax and procedures for writing and using expressions.

Chapter 5 — "Statements" describes statement labels, assignment statements, procedure reference statements, structured statements, and control statements.

Chapter 6 — "Input and Output" describes the main features of input and output in Pascal, including the file system and the various input/output statements.

¹ RT PC is a trademark of IBM Corporation.

² AIX is a trademark of IBM Corporation.

Chapter 7 — "Program Structure" describes the procedures for creating object modules.

Chapter 8 — "Built-In Procedures and Functions" describes the IBM Pascal "built-in" procedures and functions.

Appendix A — "Built-In Procedures and Function Summary" contains a list of all procedures and functions available in RT PC VS Pascal and the modes in which they may be used.

Related Publications

You may want to refer to the following IBM RT PC publications for additional information:

- *VS Pascal User's Guide*, SH23-0127, describes the procedures for compiling and running RT PC VS Pascal programs under the AIX Operating System.
- *VS Language/Operating System Interface Library*, SH23-0131, describes the system routines that can be called from FORTRAN and Pascal programs.
- *Concepts*, GC23-0784, gives an overview of the RT PC hardware, the AIX Operating System, and supporting publications.
- *Installing and Customizing the AIX Operating System*, SV21-8001, provides step-by-step instructions for installing and customizing the AIX Operating System, including instructions for adding devices to and deleting them from the system and for defining device characteristics. This book also explains how to create, delete, and change AIX and non-AIX minidisks.
- *Messages Reference*, SV21-8002, lists messages displayed by the RT PC and explains how to respond to the messages.

- *Usability Services Guide* and *Usability Services Reference*, SV21-8003, show how to create and print text files, work with directories, start application programs, and do other basic tasks.
- *Using and Managing the AIX Operating System*, SV21-8004, contains information on using AIX Operating System commands, working with the file system, developing shell procedures, and performing such system-management tasks as creating and mounting file systems, backing up the system, and repairing file-system damage.
- *AIX Operating System Commands Reference*, SV21-8005, lists and describes the AIX Operating System commands.
- *C Language Guide and Reference*, SV21-8008, provides information for writing, compiling, and running C language programs.
- *AIX Operating System Technical Reference*, SV21-8009, describes the system calls and subroutines a programmer would use to write application programs. This book also provides information about the AIX Operating System file system, special files, miscellaneous files, and the writing of device drivers.
- *AIX Operating System Programming Tools and Interfaces*, SV21-8010, describes the programming environment of the AIX Operating System and includes information about the use of operating system tools to develop, compile, and debug programs.
- *AIX Operating System DOS Services Reference*, SV21-8012, provides step-by-step information for using the AIX Operating System shell. In addition, this book describes the DOS system services.
- *User Setup Guide*, SV21-8020, provides instructions for setting up and connecting devices to system units. It also gives procedures for installing the AIX Operating System and for testing the setup.
- *Guide to Operations*, SV21-8021, describes system units, displays, console keyboard, and other devices that can be attached to the RT PC. This guide also includes procedures for operating the hardware and for moving system units.

- *Problem Determination Guide*, SV21-8022, provides instructions for running diagnostic routines for hardware and problem-determination procedures for software.

You may want to consult the following IBM publications for additional information:

- *Pascal/VS Language Reference Manual*, SH20-6168, describes the implementation of the Pascal/VS compiler.
- *Pascal/VS Programmer's Guide*, SH20-6162, describes how to compile and execute Pascal/VS programs.

Contents

Chapter 1. Introduction	1-1
Methods of Presentation	1-3
Terms and Concepts	1-4
Declarations	1-4
Statements	1-9
Lexical Constructs	1-12
Character Set	1-12
Identifiers	1-13
Numbers	1-14
String Constants	1-16
Labels	1-16
Basic Symbols	1-16
Comments	1-19
 Chapter 2. Data Types	 2-1
Types	2-1
Standard Types	2-2
Defining Data Types	2-3
Simple Types	2-4
Scalar Types	2-4
Subrange Types	2-6
Type Alfa	2-7
Type Alpha	2-7
Structured Types	2-8
Array Types	2-8
String Types	2-10
Record Types	2-11
Set Types	2-18
File Type	2-19
Pointer Types	2-21
Type Stringptr	2-22
Type Identity and Assignment Compatibility	2-23
Identical Types	2-23
Assignment-Compatible Types	2-24
Defining Constants	2-24

Predefined Constants	2-25
Hexadecimal Constants	2-26
Binary Constants	2-26
Structured Constants	2-27
Chapter 3. Variables	3-1
Declaring Variables	3-1
Predeclared Variables	3-2
Establishing Variables	3-2
Lifetime of Variables	3-3
Local Variables	3-3
Global Variables	3-3
Formal Parameters	3-3
Dynamic Variables	3-4
Static Variables	3-4
Referencing or Accessing Variables	3-4
Entire Variables	3-5
Component Variables	3-5
Pointer Referenced Variables	3-8
Chapter 4. Expressions	4-1
Operators in Expressions	4-2
Order of Evaluation in Expressions	4-2
Not Operator	4-3
Unary Operators	4-4
Multiplication Operators	4-4
Addition Operators	4-6
Relational Operators	4-7
Comparison of Scalars	4-8
Comparison of Booleans	4-9
Direct Pointer Comparison	4-9
String Comparison	4-9
Set Comparison	4-10
Non-Comparable Types	4-10
Out-of-Range Values	4-11
Compile-Time Constant Expressions	4-11
Dead Code Elimination	4-12
Chapter 5. Statements	5-1
Statement Labels	5-1
Scope Of Statement Labels	5-1

Assignment Statements	5-2
Assignments to Variables and Functions	5-2
Procedure Reference Statements	5-3
Structured Statements	5-4
Begin-End — Compound Statements	5-5
Case Statements	5-5
Empty Statements	5-7
For-Do Statements	5-8
If-Then-Else Statements	5-9
Repeat-Until Statements	5-11
While-Do Statements	5-11
With Statements	5-12
Control Statements	5-14
Assert Statements	5-14
Continue Statements	5-14
Goto Statements	5-15
Leave Statements	5-16
Chapter 6. Input and Output	6-1
The File Buffer Variable	6-1
General File Handling Routines	6-2
Close — Closes a File	6-3
Eof — Determines if End-of-File Read	6-3
Get — Gets Component from a File	6-4
Put — Replaces Component of a File	6-5
Reset — Opens an Existing File for Input	6-5
Rewrite — Opens a File for Output	6-6
Update — Opens File for Input and Output	6-6
Text File Handling Routines	6-8
Cols — Determines Current Column	6-8
Eoln — Determines if End-of-Line Read	6-9
Page — Skips to New Page	6-10
Read and Readln Procedures	6-10
Seek — Allows Random Access to Files	6-16
Termin — Opens File for Input	6-16
Termout — Opens File for Output	6-17
Write and Writeln Procedures	6-17
Chapter 7. Program Structure	7-1
Compilation Units	7-1
Compilation Unit Programs	7-2

Definitions	7-3
Program Heading	7-4
Segment Module	7-4
Scope of Identifiers	7-6
Segment Scope Rule	7-6
Block Scope Rules	7-6
Standard Files	7-7
Declarations	7-8
Label Declarations	7-8
Def/Ref Declarations	7-8
Static Declarations	7-10
Value Declarations	7-11
Space Declarations	7-12
Constant Definitions	7-15
Type Definitions	7-16
Variable Declarations	7-16
Procedure and Function Declarations	7-16
Procedure Definition	7-17
Function Definition	7-19
Parameters for Procedures and Functions	7-20
Calling Attributes	7-26
Main Attribute	7-26
Reentrant Attribute	7-27
FORTRAN Attribute	7-27
External and Forward Attributes	7-27
Chapter 8. Built-In Procedures and Functions	8-1
String Manipulation Routines	8-1
Compress — Replaces Multiple Blanks	8-3
Delete — Deletes Characters from String	8-4
Index — Returns Starting Index	8-5
Length — Determines String Length	8-6
Lpad Procedure — Pads or Truncates String on the Left	8-7
Ltrim — Removes Leading Blanks	8-7
Maxlength — Returns Maximum Length of a String	8-8
Readstr — Converts a String to Another Type	8-9
Rpad Procedure — Pads or Truncates String on the Right ...	8-11
Substr — Returns Substring	8-11
Token — Returns String as Alpha	8-13
Trim — Removes Trailing Blanks	8-14

Writestr — Converts Data into Strings	8-15
Storage Allocation Routines	8-16
Dispose — Disposes of Allocated Storage	8-17
Mark — Marks Position of Heap	8-17
New — Allocates Storage	8-18
Release — Releases Allocated Memory	8-21
Arithmetic Routines	8-23
Abs — Computes Absolute Value	8-23
Arctan — Computes Trigonometric Arctangent	8-24
Cos — Computes Trigonometric Cosine	8-24
Exp — Computes Exponential of Value	8-24
Ln — Computes Natural Logarithm of Value	8-25
Random — Computes a Random Number	8-25
Sin — Computes Trigonometric Sine	8-26
Sqr — Computes Square of a Number	8-26
Sqrt — Computes Square Root of Value	8-27
Boolean Attribute Routine	8-27
Odd — Tests an Integer for Odd or Even	8-27
Value Conversion Routines	8-28
Chr — Converts Integer to Character	8-28
Float — Converts Integer to Real	8-29
Itohs Function — Converts Integer to Hexadecimal String ..	8-29
Ord — Converts Type to Integer Value	8-30
Pack — Copies Unpacked Array to Packed Array	8-31
Round — Rounds to Nearest Integer	8-32
Scalar Conversion — Converts Integer to Scalar	8-33
Str — Converts to String	8-33
Trunc — Truncates to Nearest Integer	8-34
Unpack — Copies Packed Array to Unpacked Array	8-34
Miscellaneous Low-Level Routines	8-36
Addr — Returns Memory Location of a Variable	8-36
Sizeof — Determines Size of Data Element or Type	8-36
Control Routines	8-37
Clock — Gets Execution Time	8-38
Datetime — Gets Date and Time	8-38
Halt — Terminates a Program with Return Value	8-39
Parms — Gets Execution Parameters	8-39
Retcode — Sets Program Return Code	8-40
Return — Exits from a Procedure or Function	8-40
Additional Standard Routines	8-41
Hbound — Returns Upper Bound of an Array	8-42

Highest — Returns Highest Value of a Scalar Type	8-43
Lbound — Returns Lower Bound of an Array	8-43
Lowest — Returns Lowest Value of a Scalar Type	8-44
Max — Returns Maximum Value of Scalars	8-45
Min — Returns Minimum Value of Scalars	8-45
Picture — Formats According to Picture Value	8-46
Pred — Determines Predecessor of Value	8-48
Succ — Determines Successor of Value	8-49
 Appendix A. Built-In Procedure and Function Summary	 A-1
Index	X-1

Chapter 1. Introduction

IBM RT PC VS Pascal is an easy-to-use, fast performance, high-level programming language for the RT Personal Computer. It compiles source code in Pascal as defined by IBM Pascal VS and the ANSI-83 standard definition of Pascal.

In addition to excellent performance, RT PC VS Pascal offers these enhanced functions:

- Automated installation
- Source compatibility with IBM Pascal VS Release 2.2
- Source compatibility with ANSI X3.97-1983 definition of Pascal
- Optimized executable code
- Excellent compile-time performance
- An operating system interface library
- No significant limit on program size
- No significant limit on data size
- Separate unit compilation
- Access to command-line options
- Common development/debugging environment
- Detailed screen messages
- Easy inter-language linkages with FORTRAN and C.

You may use one of two compiling modes: IBM mode, or ANSI mode. You may work in the mode you need or with which you are most familiar.

IBM Mode

This mode lets you compile code that uses IBM Pascal VS Release 2.2 statements, data types, math libraries, and directives. When this mode is used, only IBM mode syntax programs are compiled.

You may develop and run IBM mode programs entirely on the RT PC. As a cost effective development tool, you may develop and run IBM mode programs on the RT PC as an independent workstation and then move them to a mainframe that uses Pascal VS.

You may take programs written in Pascal VS from a mainframe and run them on your RT PC.

IBM mode contains all of the ANSI-83 Standard functions; you may use ANSI-83 code that you have previously written and enhance it using the additional IBM mode functions.

ANSI Mode

This mode lets you compile code that uses only the ANSI X3.97-1983 definition of Pascal. When this mode is used, only ANSI-83 standard syntax programs are compiled.

The information describing the ANSI-83 definition of Pascal appears in black print in this manual. Extensions to the ANSI-83 definition appear in blue print.

You also have the advantage that you may mix modes in creating an executable program. However, each separate compilation unit may be only a single mode.

The IBM RT PC supports IBM Pascal VS and the ANSI-83 definition of Pascal statements, data types, math libraries, and directives except as noted. Programs using the ANSI-83 definition of Pascal may be modified to include IBM mode extensions.

You should note that some programs may produce different results when run on the RT PC compared to other machines because of differences in machine architecture, the operating system, or compiler implementation. These differences are noted where applicable.

This reference manual describes all of the standard features of Pascal as well as the enhanced functions and capabilities incorporated into the IBM RT PC VS Pascal. You should have a knowledge of Pascal concepts and some experience writing Pascal programs. If you do not have this basic knowledge and experience, you may wish to obtain one of the tutorial-style Pascal texts which are commercially available.

Methods of Presentation

The notations used to explain the RT PC VS Pascal language syntax in this manual are as follows:

- **Bold** letters and words appear as they should in programs.
- *Italicized* letters and words indicate that data objects should be substituted in their place in actual program statements.
- An ellipsis .. meaning "through", indicates an ordered sequence where only the start and end elements are specified.
- Brackets [and] indicate optional items and subscripts of an array.
- Braces { and } enclose elements that can be repeated "zero to many times".

Note: Although braces are also one of the forms of comment delimiters in RT PC VS Pascal, this should not cause any difficulty. The one case where misinterpretation might occur is in the definition of comments, and this is explicitly pointed out.

Pascal standard information appears in black print while the extensions and enhancements of the IBM mode appear in blue print.

Occasionally, it is necessary to use a specific character which results from depressing two keys at the same time. For example, the RT PC VS Pascal **eof** character is represented in this manual as **Control-D**. To enter this character in a file, both the Ctrl and the D keys must be depressed simultaneously.

Terms and Concepts

This section defines RT PC VS Pascal terms and concepts as they are implemented on the IBM RT PC.

An RT PC VS Pascal program consists of a series of declarations and statements. "Declarations" serve to define program objects. "Statements" determine the actions that are performed upon the program objects. These two items, declarations and statements, serve to describe a computer program.

Declarations

"Definable objects" in RT PC VS Pascal include variables, functions, procedures, and files. Each of these object types requires an "identifier" and usually a "type description". An object's identifier serves to identify that object so that it can be referenced later. An object's type description defines its operational characteristics and, in some cases, indicates a reference notation. It is important to note that all user-defined objects must be fully described, especially as to their type. RT PC VS Pascal has an advantage over many other programming languages in that it does not supply any default attributes for undeclared identifiers. RT PC VS Pascal has an additional advantage in that the traditional ordering of these definable objects is not required.

One of RT PC VS Pascal's strong points is the ability to define new types. RT PC VS Pascal supplies a number of predefined or basic types, such as integer, char, etc. In addition, RT PC VS Pascal supplies notations for defining new (user-defined) types, both in terms of the basic types and in terms of other user-defined types. A type can be described directly in a declaration. It can also be referenced by a type identifier if that type identifier is defined in another type declaration.

An RT PC VS Pascal object can execute functions or operations only within an area indicated by its type. For example, most binary operators can perform operations only on objects of the same type. This means that characters and integers cannot be added directly. These operational constraints are fixed, as are the rules for type identity and assignment compatibility.

Departures from the rules must be spelled out explicitly in terms of conversion functions.

The basic data type is the "scalar type", often referred to as an enumerated type. A scalar definition indicates an ordered set of values with each identifier in the set standing for a specific value.

In addition to the definable scalar types, there are standard basic types: integer, char, real, [shortreal](#), [alpha](#), [alfa](#), text, pointer, [stringptr](#), and boolean. With the exception of the boolean type, the values for these standard types are denoted by numbers or quoted characters. A type may also be defined as a subrange of a scalar type by indicating the lower and upper bounds of the subrange.

"Structured types" are collections of various data types. They are defined by describing the types of the components and by indicating the structuring method. The structuring methods differ in the way that components of a structured variable are selected and the operations in which they can participate. Pascal provides certain basic ways to construct a structured type: array, record, set, [string](#), and file.

An "array" is useful for keeping a list of data when each item in the list is of the same type. It consists of elements or components which are all of the same type. A component is selected by a computable index. The type of such an index must be a scalar, and is determined at the time the array is declared. For example, an array with four integer components is defined as:

```
type barray = array [1..4] of integer;
```

A "record" is made up of parts, or fields, of related data that may not be of the same type. A unique identifier may be associated with each field selector to allow access to that field and, therefore, to any field in the record. Each field, then, has its own type and its own identifier. Unlike an array element index, a field selector is not a computable quantity. The field selectors are defined at the same time the record is defined.

A record type may contain two parts: a fixed part and a variant part. The "fixed" part of the record contains field structures that are not subject to change. The "variant" part, however, is used to create a field whose structure (or layout) may vary between different instances of the record. A "tag field" is used to specify which variant structure is valid. The actual value of the tag field then acts as a selector to the appropriate variant. This allows a

single record structure to hold different information as needed in different instances, without unnecessary duplication of fields.

```
type
emp_class = (full_time, part_time, contractor);
personnel = record
    first_name      : string (20);
    last_name       : string (20);
    mid_init        : char;
    case class      : emp_class of {variant part}
        full_time   : (salary: real;
                        retirement_plan: integer;
                        dependents: integer;
                        credit_member: boolean);
        part_time    : (wage: real;
                        max_hours: real);
        contractor   : (fee: real;
                        job_class: integer;
                        job_hours: real);
    end;
```

In the above example, the fixed part of the record is made up of the `first_name`, `last_name`, and `mid_init` fields. The variant part begins with the statement `case class: emp_class of`. The field `class` is the tag field used to specify which of the variants (`full_time`, `part_time`, or `contractor`) is valid. The value of class must first be defined before accessing any of the variant fields.

An assignment to the field `employee.class` must be made before assigning any values to the subsequent variant fields. It is an error to access any variant fields that are inconsistent with the current value of the tag field.

The programs in the following illustrations use the record type named `personnel` defined in the previous example.

```
var
    employee : personnel;
```

In this instance, the assignments:

```
employee.class := part_time;
employee.wage  := 9.67;
employee.max_hours := 20;
```

are correct for the variant `part_time` specified.

The following assignments would constitute an error:

```
employee.class      := contractor;
employee.salary     := 32,767.00 {no such field in the
                                   contractor variant}
employee.max_hours  := 40;        {no such field in the
                                   contractor variant}
```

or

```
if employee.class = full_time then
  employee.wage := employee.wage + .75; {no wage field in
                                         fulltime variant}
```

The tag field is an actual field of the record, and, like any other field, it occupies space in the record. The tag field identifier may be omitted, in which case the above record could be defined as:

```
type
emp_class = (full_time, part_time, contractor);
personnel = record
  first_name: string(20);
  last_name  : string(20);
  mid_init   : char;
  case emp_class of {variant part,
                    tag field omitted}
    full_time : (salary: real;
                 retirement_plan: integer;
                 dependents: integer;
                 credit_member: boolean);
    part_time  : (wage: real;
                 max_hours: real);
    contractor: (fee: real;
                 job_class: integer;
                 job_hours: real);
  end;
```

In this record definition, only the types by which the variants are selected are specified. The variant fields can be accessed without referencing a tag field value. If the tag field is not specified in the variant portion of the record, it is necessary to ensure that only valid variant fields are accessed.

Since no tag field exists in this case, no space is allocated in the record for a tag field value.

A "set" is a collection of elements of the same base type. It might be a user-defined scalar type or a subrange of some scalar type such as integer or char. In this example, "varieties" is of type set, and "fruit" is the base type.

```
program fruit_groups;
type fruit      = (orange, lemon, lime, apple,
                  pear, blueberry, raspberry);
   varieties    = set of fruit;
var citrus, berries, other : varieties;
begin
  citrus := [orange, lemon, lime];
  berries := [blueberry, raspberry];
  other  := [apple, pear];
end.
```

A "string data type" is a sequence of characters whose length can vary dynamically during program execution. A string has a maximum length (the static length) that is determined when it is defined. There are a variety of procedures and functions that can be used to manipulate strings. An example of a string definition is:

```
city   : string(20);
state  : string(2);
```

A "file" is a sequence of records of the same type. The sequence is normally associated with external storage or input and output devices. Files allow an RT PC VS Pascal program to transfer data to and from these devices. A file can be either sequential or random. A sequential file has a natural ordering with records of the file being accessed one after the other; a random file allows access to any given record of the file directly without regard to sequence. Examples of file type definitions are:

```
type f1 = file of personnel;
type f2 = file of integer;
type f3 = file of char;
```

"Explicitly declared variables" are considered static because their characteristics are known at compile time and do not change. A declaration associates an identifier with the variable. The identifier can then be used to refer to that variable.

Variables can also be dynamically allocated by executable statements. Dynamic allocation of a variable produces a pointer that substitutes for an explicit declaration. This pointer is subsequently used to refer to the dynamically-allocated variable. A pointer variable may only assume values pointing to variables of a specific type: the pointer is bound to that type. However, it may be assigned to other pointer variables of the same type. Any pointer can assume the value `nil`, which is a universal pointer that is not bound to a specific type. The following example illustrates a dynamic variable.

```
type ptr      = @folder;
  folder= record
      name      : string(30);
      gpa       : real;
      grade     : 1..12;
      next      : ptr;
  end;
```

Statements

The "assignment statement" is the fundamental RT PC VS Pascal statement. It assigns a new value to a variable or a component of a variable. New values are obtained by evaluating expressions. Expressions consist of variables, constants, sets, operators, and functions operating on specified objects to produce new values. Operands of expressions are either declared in the program or are standard RT PC VS Pascal entities. RT PC VS Pascal defines a fixed group of operators that can be used to assign new values to variables. These groups of operators are: arithmetic, boolean, set, and relational. Examples of assignment statements are:

```
a  := b;
c[i] := 12 * 4 + (b * a);
```

A "procedure statement" is used to execute a designated procedure. This is known as activating or calling the procedure. The procedure statement contains the procedure identifier and, optionally, a list of optional parameters, if any are defined. If the procedure `do_it` is declared as:

```
procedure do_it;
begin
  writeln ('did it.');
```

```
end;
```


the procedure statement `do_it` executes the procedure.

Assignment and procedure statements are the basic elements of structured statements. Structured statements specify the sequential, selective, or repetitive execution of their component statements. Sequential execution is obtained by the compound statement using **begin** and **end**; conditional and selective execution by the **if** and **case** statements; and repetitive execution by the **while**, **repeat**, and **for** statements.

A structured statement can be given a name (identifier) and, subsequently, be referenced by that name. The statement is then considered a procedure and its declaration is a procedure declaration. The procedure declaration can contain type declarations, variable declarations, and further procedure declarations. These subsequent declarations can be referenced only within that procedure and are, therefore, local to the procedure.

The "scope" of an identifier is the area of the program or procedure in which the identifier can be referenced. The scope of the identifier is then local to that procedure or program. Procedures may be nested; therefore, the scope of the identifiers may be nested. Objects declared in the main program block that are not local to any procedure are referred to as global because their scope is that of the entire program.

A procedure may include a list of parameters. Each parameter is indicated by an identifier known as the formal parameter. A formal parameter is a variable identifier used in the procedure. A procedure statement may include a list of actual parameters. These parameters are passed to the procedure when it is called. An actual parameter might be a variable, constant, expression, procedure, or function. This example contains a procedure declaration heading that includes a list of formal value parameters:

```
procedure show (x : integer; y, z : real);
```

The following statement can then be used to pass the actual parameters:

```
show (36, 1.4, W);
```

There are three types of parameters: value parameters, variable parameters, and procedure or function parameters. A "value parameter" is an actual parameter that is evaluated once. The formal parameter then represents a local variable initialized to the value of the actual parameter. A "variable parameter" is an actual parameter that is a variable. The formal parameter

actually references and can alter that variable. Possible array indexes are evaluated before activation of the procedure or function. A "procedure or function parameter" is an actual parameter that is a procedure or function identifier.

Functions are declared the same way as procedures. The difference is that a function returns a value. A function performs an operation and computes a single value. Functions can return any type except file. The type of the returned value must be specified as part of the function declaration. A function reference must appear in the context of an expression.

This program demonstrates the use of functions:

```
program show;
var i, j, result : real;
  function operation (x, y : real; op : char) : real;
    function add (a1, a2 : real) : real;
      begin
        add := a1 + a2;
      end;

    function subtract (s1, s2 : real) : real;
      begin
        subtract := s1 - s2;
      end;

    function mult (m1, m2 : real) : real;
      begin
        mult := m1 * m2;
      end;
    begin
      case op of
        '+' : operation := add (x,y);
        '-' : operation := subtract (x,y);
        '*' : operation := mult (x,y);
      end;
    end;
end;

begin
  i := 6.78;
  j := 8.45;
  result := operation(i,j, '*');
  write (result);
end.
```

RT PC VS Pascal procedures and functions may be recursive. This means that a procedure or function can call itself before the current activity is complete. On each activation, a fresh set of local data is created. Recursive activation can be direct, with the reference contained within the procedure or function itself; or it can be indirect, with the reference coming from another procedure or function that is referenced from the current procedure or function.

Lexical Constructs

The RT PC VS Pascal language constructs (identifiers, basic symbols, and constants) are composed of one or more consecutive elements of the following:

- character sets
- identifiers
- numbers
- string constants
- labels
- basic symbols

Character Set

RT PC VS Pascal uses an extended form of the ASCII character set for all text-related processing. ASCII stands for the American Standard Code for Information Interchange. There are 128 characters in the ASCII character set: 52 letters (uppercase and lowercase A - Z), digits 0 - 9, the space (blank) character, 33 control codes (such as carriage return and line feed), and 32 graphic characters (such as the colon and equal sign). RT PC VS Pascal also allows an additional 128 values to be used as data values for a total of 256 possible data values. The character sets recognized by the RT PC VS Pascal compiler are as follows:

A letter may be any of the following characters.

```
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m
n o p q r s t u v w x y z $
```

A digit may be any of the following characters.

```
0 1 2 3 4 5 6 7 8 9
```

A hexadecimal digit may be any of the following characters.

```
digit A B C D E F
digit a b c d e f
```

The ASCII graphic characters may be any the following characters.

```
! " # $ % & ' ( ) * +
, - . / : ; < = > ? @
[ \ _ ] ^ _ { | } ~
```

Note: The dollar sign (\$) is defined as a letter in IBM mode.

Identifiers

RT PC VS Pascal identifiers are used to point out constants, variables, procedures, and other language objects. An identifier must start with a letter or a dollar sign (\$). It may contain any combination of letters, digits, the dollar sign, and the underscore character. The underscore is frequently used to mark off spaces in the identifier so that it is more readable and meaningful.

An identifier may have any length, but only the first 16 characters are significant to the compiler. Both upper and lowercase letters are "folded" into a single case in the compiler, which makes them equivalent.

Examples:

```
$here_and_there   August_1979   Steve_and_Jeff  
Tau_Epsilon_Xi   DragonsEgg    up$AND$down  
UPanddown        upandDOWN    upANDdown
```

The three identifiers of the last line in the examples are equivalent because the compiler folds letters into a single case.

Invalid Examples:

```
1st_character_must_be_a_letter  
mustn't_have_odd_#"[_characters_in_it
```

Numbers

Numbers are used to represent integer, real, and **shortreal** data elements. Integers are assumed to be decimal numbers unless designated as hexadecimal.

Unsigned Integer

digit {digit}

Unsigned Real

integer.integer
or
integer.integerEscale-factor
or
integerEscale-factor

E

designates values represented in scientific or exponential form. The E precedes the exponent.

scale-factor

is either a signed or unsigned integer. The sign can be either a "+" or "-".

Unsigned Number

integer
or
real

Examples:

667	unsigned decimal integer
+99 -457	signed decimal numbers
\$3e8	a hexadecimal number
0.0	the real number zero
3.14159	unsigned real number

Invalid Examples:

5.	no digit after the point
.618	no digit before the point
5.E10	no digit after the point
2FC9	invalid decimal number
F034	an identifier, not a hex number

String Constants

Sequences of characters enclosed in apostrophes are called strings. Strings of one character are constants of type `char`. A string of '*n*' characters, where '*n*' is greater than one, is either a string value or a value of the type packed array `[1..n]` of `char`. The exact type of such a string constant is determined from the context in which it appears.

A string constant which is simply two consecutive apostrophes (') represents a variable string constant of length zero.

Labels

A label is an unsigned integer or an **identifier** used to mark statements as the potential target of a **goto** statement. Labels are **identifiers** or unsigned integer constants in the range 0..9999. For a description of identifiers, see "Identifiers."

Basic Symbols

Pascal has a set of "basic symbols" which the compiler uses for specific purposes in the language. These basic symbols include selected identifiers (reserved words), graphic characters, and pairs of graphic characters. Basic symbols are used as keywords, operators, delimiters, and separators and appear throughout this manual. Figure 1-1 contains the list of RT PC VS Pascal reserved words and Figure 1-2 contains the special symbols used in RT PC VS Pascal.

Note: No RT PC VS Pascal reserved words may be used as user-defined identifiers.

and	end	of	space
array	file	or	static
assert	for	otherwise	then
begin	function	packed	to
case	goto	procedure	type
const	if	program	unit
continue	in	range	until
def	label	record	value
div	leave	ref	var
do	mod	repeat	while
downto	nil	return	with
else	not	set	xor

Figure 1-1. Reserved Words

+	addition operator
-	subtraction operator
*	multiplication operator
/	division operator (for real and shortreal data types)
:=	assignment operator
.	terminates a RT PC VS Pascal compilation unit; separates integer from fraction in a real or shortreal number; indicates reference to a field of a record
,	separates items in lists
;	statement and declaration separator

Figure 1-2 (Part 1 of 3). Special Symbols

:	used after case labels, statement labels, variable and parameter definitions
=	relational equality operator; used in constant and type definitions
<> ~ =	relational operators for inequality
<	relational operator for "less than"
<=	relational operator for "less than or equal to"
>=	relational operator for "greater than or equal to"
>	relational operator for "greater than"
(and)	enclose lists of elements; enclose parts of expressions that are considered indivisible factors
[and]	enclose array subscripts and lists of set elements
(. and .)	enclose array subscripts and lists of set elements (alternate form)
{ and }	comment delimiters
(* and *)	comment delimiters
/* and */	comment delimiters
>>	right logical shift on integer
<<	left logical shift on integer
	catenation operator
&&	boolean xor operator, logical xor on integer and set exclusive union

Figure 1-2 (Part 2 of 3). Special Symbols

@ ->	pointer reference operator
&	boolean/logical and
~	boolean not, one's complement on integer or set complement
	boolean or, logical or on integer
..	subrange notation
'	used to begin and end string constants

Figure 1-2 (Part 3 of 3). Special Symbols

Spaces (blanks) are used to separate RT PC VS Pascal constructs. Identifiers, reserved words, and constants must not join each other and cannot contain embedded spaces. Multiple-character basic symbols such as <= must not contain embedded spaces.

Otherwise, spaces may and should be used freely to improve program readability. Outside of character and string constants, where a space represents itself, spaces have no effect.

Comments

Comments in RT PC VS Pascal may appear anywhere that a space may appear and, in fact, serve the same purpose as spaces. Note that a comment delimiter within a string constant is part of the string constant. It is not interpreted as a comment delimiter. Comments can span multiple lines, thereby providing a "block comment" capability.

RT PC VS Pascal comments are enclosed between braces, { and }, between the characters (* and *), or between the characters /* and */. A comment may include any printable character except the character(s) that act as the delimiter. If a comment is begun with { it must end with a }; if it is begun with (* it must end with a *).

Examples:

```
{ comment enclosed in braces }
```

```
(* comment within other comment delimiters *)
```

```
{ To close this comment, *) does not work.  
  The comment continues until it finds a }
```

```
/* comment enclosed in alternate delimiter form */
```

Chapter 2. Data Types

A major advantage of the RT PC VS Pascal language is the ease with which data can be described and manipulated. This ability is provided in RT PC VS Pascal through the notion of a data type or simply "type". A type specifies the characteristics of data. Each variable and constant used in a program has a type associated with it.

RT PC VS Pascal provides predefined types or "standard types", which are described in this chapter. Also included is an explanation of user-defined types, structured types, and pointer types as well as the assignment capability of types. The procedures for using predefined and structured constants are also described.

Types

A "type" defines a set of values that a variable, constant, or expression may take. A type has an associated size, but reserves no storage space itself. Storage is only reserved when a variable is declared as an instance of that type. Although RT PC VS Pascal data types can be quite complex, they are ultimately composed of simple unstructured components.

In addition to having a size and a set of values associated with a given type, a collection of operations can be performed on all values of that particular type.

Standard Types

RT PC VS Pascal provides the following predefined standard types:

integer represents an implementation-defined subset of the integers. It is equivalent to a subrange defined by the type definition:

```
integer = -2147483648..2147483647
```

The integer data type occupies 32 bits of data storage.

real represents floating-point data. Variables of this type occupy 8 bytes of memory and align on a double word boundary. All real arithmetic is done using double precision floating-point instructions.

shortreal represents floating-point data. Variables of this type occupy 4 bytes of memory and align on a word boundary. All shortreal arithmetic is done with single precision floating-point instructions. Double precision floating-point instructions can be used to perform operations between data of types real and shortreal. A shortreal can be passed only as an operand to a function or procedure that expects its parameter to be of type real when the parameter-passing mechanism for that parameter is a value or constant.

A shortreal is converted to a real when one operand of a binary operation is a shortreal and the other is a real. To convert a shortreal to a real, assign a shortreal to a real variable.

Notes on Real and Shortreal

The following points apply to both real and shortreal.

- There is no subrange.
- They cannot be used to index an array.

- The following predefined functions are not defined and, therefore, cannot be used with shortreal or real.

succ
 pred
 ord
 highest
 lowest

boolean	represents the ordered set of truth values whose constant values are false and true. Boolean is conceptually equivalent to an ordinal type specified by the type definition, i.e., boolean = (false, true). Variables of this type occupy 1 byte.
char	is a scalar type that consists of all the values of the ASCII character set. Values of this type occupy 1 byte and are aligned on a byte boundary.
text	is defined as a file of char. In addition to the predefined procedures which perform input and output, there are several defined procedures which operate only on files of type text. These procedures convert characters to internal representation (ASCII) and allow control of the output field lengths. These procedures include: get , put , eof , eoln , reset , and rewrite .

Defining Data Types

RT PC VS Pascal data types are used to define sets of values that Pascal variables may assume and, in many cases, to define a notation for referencing such variables. RT PC VS Pascal provides a small number of standard types, reserved identifiers for these types, and notations for defining new types in terms of existing types. See "Standard Types."

Type declarations introduce (user-defined) types and identifiers for those newly declared types. A data type is defined as:

type-identifier = Pascal type;

Type declarations can be used for purposes of brevity, clarity, and accuracy. Once declared, a type may be referred to anywhere in the program by its declared type-identifier.

Simple Types

Simple types are those that have neither structure nor components. The simple types that are explained in detail in the following sections are:

scalar
subrange
alfa
alpha.

Scalar Types

A scalar type defines a well-ordered set of values by enumerating the identifiers that denote those values. A scalar type is also known as an enumerated type or an ordinal type. An ordinal type is represented by the ordered set of integers 0, 1, 2, 3, and so on, with the first identifier being 0, up to the last identifier which is $n-1$ (where n is the number of identifiers in the list).

A scalar type is defined as:

(identifier {,identifier})

Examples:

```
salad_greens = (spinach, lettuce, collards);  
bottle_sizes = (pint, quart, gallon);  
mealtimes = (breakfast, lunch, dinner);
```

Scalar Conversions

RT PC VS Pascal predefines the **ord** function to convert any scalar value into a 32-bit integer. The scalar conversion functions convert an integer into a specified scalar type. An integer expression is converted to another scalar type by enclosing the expression in parentheses and prefixing it with the type identifier of the scalar type. A subrange error occurs if the operand is not in the range `0..ord(highest(scalar type))`. The conversion is performed as the inverse of the **ord** function.

The definition of any type identifier that specifies a scalar type, either enumerated scalars or subranges, forms a scalar conversion function. By definition, the expression `char(x)` is equivalent to `chr(x)`; `integer(x)` is equivalent to `x`; and `ord(type(x))` is equivalent to `x`.

Examples:

```
type  
  week = (sun,mon,tue,wed,thur,fri,sat);  
var  
  day : week;  
  .  
  .  
{The following assigns sat to day}  
  day := week(6);
```


Subrange Types

A "subrange type" represents a subrange of values of another scalar type. It is defined by a subrange type identifier or by a lower and an upper bound. The lower bound must not be greater than the upper bound, and both bounds must be of identical signed scalar types. A subrange type is defined by indicating the lower and upper bounds of the values in the subrange, separated by a ".." symbol.

If the reserved word **range** is used in the subrange definition, the minimum value may be any expression that can be computed at compile time. If range is not used, the minimum value of the subrange must be a constant.

subrange-type = lower .. upper

Values from a subrange and values from the parent range or another subrange of the parent range can be assigned to each other and can enter into assignment, comparison, and other binary operations.

Examples:

```
small_integer = 0..15;  
days_in_year = 1..366;  
positive_integer = 0..32767;  
lower_case_letters = 'a'..'z';
```

```
colors = (red, orange, yellow, green, blue);  
hot_colors = red..yellow;  
cold_colors = green..blue;  
hues = red..blue;
```

```
days = (sat, sun, mon, tues, wed, thurs, fri);  
weekdays = mon..fri;  
weekends = sat..sun;
```

```
codes=range chr(0)..chr(255);
```

Note: The lower bound of a subrange definition that is not prefixed with **range** must be a simple constant.

Type Alfa

The standard type **alfa** is defined as:

```
const
  alfalen = 8;
type
  alfa = packed array[1..alfalen] of char;
```

Any packed array[1..n] of char, including **alfa** may be converted to type string by the predefined function **str**.

Type Alpha

The standard type **alpha** is defined as:

```
const
  alfalen = 16;
type
  alpha = packed array[1..alfalen] of char;
```

Any packed array[1..n] of char, including **alpha** may be converted to type string by the predefined **str** function.

Structured Types

"Structured types" represent collections of objects. They facilitate the definition of new and larger types based upon other existing types as components. Structured types are defined by describing their element types and indicating a structuring method.

The following structuring methods are available:

- array
- string
- record
- set
- file

These differ in the accessing mechanisms and in the notations used to select elements from the collection. Each of the methods is described in the following sections.

A structured type may be given the packed storage attribute. This tells the compiler that the structure is to use data storage economically by packing the components of the structure densely. Packing is often achieved at a cost of larger code size and slower execution speed.

Array Types

An "array type" is a structure consisting of a fixed number of components that are all of the same type (known as the component type). Array elements are designated by indexes, which are values belonging to the index type. The array type-definition specifies the component type as well as the index type. An array type consists of the symbol **array** followed by one or more *index-types* which are separated by commas and enclosed in square brackets as shown in the examples.

An array is defined as:

array [*index-list*] of *type*

index-list

is one or more simple types separated by commas.

If *n* index-types are specified, the array is an *n*-dimensional array. Note that the definition for an array type means that there are two ways of specifying an array. By definition, a component of an array can be another array type. Thus a three-dimensional array could be specified as:

```
local = array [1..10, 11..20, 21..30] of phone;
```

```
longdist = array [1..10] of array [11..20]  
            of array [21..30] of phone;
```

The alternative forms of specifying array types are equivalent. The first form can be thought of as a shorthand notation for the second form. There is a similar choice of notations when specifying the index elements for accessing an array component.

When the index type is a subrange of the type integer, the type `packed array [1..n] of char` is a special case. Objects of this type can be compared as single entities up to a maximum length of 32767 characters whereas arrays of other data types must be compared element by element. A literal string constant can be assigned to a packed array of char when the length of the literal string is less than or equal to the declared length of the packed array of char. The type of a literal string of length *n* (where *n* is greater than 1) is compatible with `packed array [1..n] of char`.

Examples:

```
rows = 1..3;  
columns = 1..4;  
  
bottle_quantities = array [bottle_sizes] of integer;  
  
standard_case = packed array [rows] of array [columns] of bottles;  
  
token = packed array [1..100] of char;
```

String Types

A string is defined as a packed array [1..n] of char whose length varies at execution time up to a compile-time specified maximum. The length of the array is obtained during execution by the **length** function. The length is managed implicitly by the operators and functions which apply to strings during execution by the **maxlength** function. The length of a string variable is determined when the variable is assigned. By definition, string constants belong to the type string.

A string variable is defined as:

string(static-length)

static-length

is an integer constant in the range 1..32767

The string constant '' (two adjoining apostrophes) represents a null or zero-length string.

Examples:

```
manila = string(100);  
punched_card = string(80);
```

A string variable may be subscripted with an integer expression to reference individual characters. A subscript of 1 references the first character. The subscript value must not be less than 1 and must not exceed the string's length.

The constant expression that follows the string qualifier in the type definition is the maximum length that the string may obtain. It must be in the range of "1..32767".

Any variable of a string type is compatible with any other variable of a string type. The maximum length of a string has no bearing on type compatibility.

Implicit conversion is performed when assigning a string to a variable whose type is `packed array[1..n] of char`. All other conversions must be done explicitly. The assignment of one string to another may cause a run-time error if the actual length of the source string is greater than the maximum length of the target.

A string can be converted to a `packed array[1..n] of char` on assignment. A string should be the same length or shorter than the array to which it is assigned. If it is shorter than the array, blanks appear on the right. If it is longer than the array, a run-time error results.

A string being passed by value or `const` to a formal parameter that requires a `packed array[1..n] of char` is converted. If it is shorter than the array, blanks appear on the right. If it is longer than the array, a run-time error results.

Record Types

A "record type" is a structure consisting of a fixed number of fields or components. These components may be of different types. The definition specifies a type and an identifier which is used to reference the field or component of the record. The scope of these field identifiers is the definition of the record itself. This means that the same field identifier may be used in more than one record. A field identifier is only accessible when referring to a variable of this record type.

Record components that are themselves records do not inherit the packing attributes of the main or parent record. Each component that is a record has independent packing attributes.

A field of a record need not be named; the field identifier has a type and length but no identifier. In this case, the field serves only as padding: it cannot be referenced.

Example:

```
rec = record
a,
b : integer;
  : char;
c : char;
end;
```

A record type is defined as:

record *field-list* **end**

field-list

is one of the following:

fixed-part
fixed-part ; variant-part
variant-part

A fixed-part is defined as:

record-section {; *record-section*}

A record-section is defined as:

field-identifier-list : type or type

field-identifier-list

is a list of one or more identifiers separated by commas.

Using this syntax, a "record-type" consists of the symbols **record** and **end** enclosing a *fixed-part*, which is a list of one or more *record-sections* separated by semicolons. Each *record-section* introduces a list of one or more field identifiers which are separated by commas and a specification of their type. The term identifier can be used in place of the term field-identifier.

A "variant record" consists of two portions: one portion which is always the same and one portion whose layouts change in chosen instances (variations). The specific variant that is selected in any given instance is determined by an optional tag field. Such a structure is called a variant record or a discriminated union. The tag field is often called a discriminant.

A variant-part is defined as:

case tag-field { : *type-identifier* } **of** *variant-list*

tag-field

contains a value that indicates which variant the record assumes at a given time. The tag-field is a field in the fixed part of the record. When it is followed by a type identifier, the tag-field defines a new field within the record.

The tag field's value indicates which variant the record assumes at a given time. Each variant structure is identified by a case label that is a constant of the tag field's type. Referencing a field of a variant that is inconsistent with the tag field's value may produce unpredictable results.

type-identifier

specifies a previously defined type. The values of this type specify the possible alternative structures of the variants.

If the type identifier is missing, the tag field name must be one which was previously defined within the record. This allows the tag field to be placed anywhere in the fixed part of the record.

variant-list

is made up of a *case-label-list*. This *case-label-list* is defined as:

case-label-list : (*field-list*)

case-label-list

is a statement containing a constant or series of constants.

Note: A *type-identifier* is optional in a variant record definition. If the *type-identifier* is omitted, the *tag-field* name must be a name that was previously defined within a record.

Examples:

```
{ This example shows an ordinary record named      }
{ complexnumber that contains two fields:  a        }
{ real part and an imaginary part.          }      }
```

```
complexnumber = record
  realpart: real;
  imaginary: real;
end;
```



```

{ This example shows a variant record type      }
{ that has different sections that are accessed }
{ depending on the tags.                       }
{ First an enumerated type that is used as the  }
{ variant case selector is defined.            }

```

```

shapes = (rectangle, triangle, circle, polygon);

```

```

angle = -180..+180;

```

```

positionrec = record
  x_position: real;
  y_position: real;
  case whatshape: shapes of
    rectangle: (recbase: real;
               recheight: real);
    triangle: (tribase: real;
              triheight: real;
              triskew: angle);
    circle: (cirradius: real);
    polygon: (sidecount: integer;
             polradius: real);
  end;

```

A variant part of a record need not have a tag field at all. In this case, only a type identifier is specified in the case construct. The variant fields are still referenced by their names, but it is necessary to keep track of which variant is active during execution. Tag fields may be defined in any of these ways:

- case *i* : *integer* of

results in *i* being a tag field of type integer

- case *integer* of

means no tag field is present. The variants are denoted by integer values in the variant declaration as shown in the examples.

- case *i*: of

means that *i* is the tag field, and it must have been declared in the fixed part. The type *i* is as given in the field definition of *i* as shown in the following examples.

Examples

```
type
  shape = (triangle, rectangle),
coordinates =
  { fixed part: }
record
  x,y : real;
  area : real;
  case s : shape of
    {variant part:}
  triangle : (side : real;
              base : real);
  rectangle : (sidea; sideb : real);
end;
```

The record defined as `coordinates` contains a variant part. The tag field is `s`, its type is `shape`, and its value, `triangle` or `rectangle`, indicates which variant is in effect. The fields `side` and `sidea` occupy the same offset within the record.

```
coordinates =
  record
    s : shape;
    x,y : real;
    area : real;
    case s : of
      {variant part:}
    triangle : (side : real;
                base : real);
    rectangle : (sidea, sideb : real);
  end;

{ Record with back reference tag field }
{ The tag field was declared in the    }
{ fixed part of the record.            }
```

```

coordinates =
  record
    x,y  : real;
    area : real;
    case shape of
      {variant part:}
      triangle : (side : real;
                  base : real);
      rectangle : (sidea, sideb : real);
    end;

    { Record with no tag field }

```

Packed Records

The fields in a record are normally assigned offsets sequentially, padding where necessary for boundary alignment. In packed records, however, no such padding is done. This may save storage within the records, but it may also degrade the performance of the program. Fields of a packed record may be passed as **var** parameters to a routine.

Offset Qualifications of Fields

RT PC VS Pascal provides a method of forcing the fields of a record to begin at a specified byte offset in the record. A field name may be followed by an integer constant expression enclosed in parentheses. This expression represents the byte offset within the record that the field is to represent. The fields that are specified in this manner must be in consecutive order according to the offsets. If the offset is not specified, the field is assigned the next offset that is required for boundary alignment. If an offset specification attempts to assign an incorrect boundary for a field and the record is not packed, a compile time error occurs.

As an example of offset qualified fields within a record, consider a large control block of 100 bytes in which four fields at various offsets need to be referenced.

<u>Byte</u> <u>Displacement</u>	<u>Information</u>
0	field a (integer)
36	field b (8 chars)
80	field c (4 flags)
92	field e (integer)

The control block might be represented as:

```

type
  flags = set of
    (f1, f2, f3, f40);
  padding = packed array [1..4] of char;

  cb = packed record
    a      : integer;
    b(36)  : alpha;
    c(80)  : flags;
    e(92)  : integer;
           : padding
  end;

var
  block : cb;

{ A record with offset qualified fields }
```

Note: Offset qualifiers cannot be used on the variant part tag field. To ensure that the tag field begins at a certain offset, use the back reference tag field method described earlier. The last identifier of the fixed part should have the same name as the tag field. The offset qualifier must be put on this last identifier.

Set Types

A "set type" definition defines the base type that the set is to use in future manipulations. Sets are limited to 256 elements. The set elements must be within the range 0..255.

A set type is defined as:

set of simple-type

Examples:

```
salad_base = set of salad_greens;  
dressings = set of salad_dressings;  
lower_case = set of 'a'..'z';
```

File Type

A file type defines a sequence of elements. A file is usually associated with external storage devices or communication devices. RT PC VS Pascal supports an interactive file type, which is more suitable for terminals, in addition to the standard RT PC VS Pascal typed files and untyped files.

When a file variable f with components of type "T" is declared, there is an additional implied declaration of a so-called buffer variable or "window", also of type "T". This window is referenced by the notation $f@$ (where f is the file variable). This window is used in conjunction with the **get** and **put** procedures, which are explained in Chapter 6, "Input and Output." This window is used to append components when writing to the file and to access the components when reading from the file.

A file type is defined as:

file of type

type

specifies the component values of the file which may be unstructured values, or structured values such as arrays, records, or sets.

A file of the predefined type `text` is defined by the following type definition:

```
text = packed file of char;
```

Such a file is special because the range of its components (characters) are extended to include an end-of-line marker. This file can then be conveniently structured into lines. The `eoln` predicate described in Chapter 8, "Built-In Procedures and Functions," explains how the end-of-line is detected.

RT PC VS Pascal also supports an interactive file type that produces different results in the `read`, `readln`, and `reset` procedures. The differences are described in Chapter 6, "Input and Output." An interactive file is more suitable for use with interactive terminals.

Examples:

```
block_access = file of blocks;  
numbers     = file of integer;  
capping_line = file of bottles;  
terminal    = interactive;  
legible_file = text;
```


Pointer Types

Explicitly declared variables are accessed by referencing the identifier used to declare them. Such variables are accessible during the activation of the procedure in which they are declared. These variables are static.

Variables may also be allocated dynamically; that is, without an explicit variable declaration. These dynamic variables are created by the **new** procedure. Because these variables do not have an associated identifier, they are accessed by a pointer value that is generated when the variable is allocated. A pointer type is, therefore, a value that points to a variable of a specific type.

A pointer type is defined as:

pointer-type = @*type-identifier*
or
pointer-type = ->*type-identifier*

There is a universal pointer value known as "nil", which belongs to any pointer type. It represents a pointer that points to no element.

Examples:

```
type
ptr = @ element;
element = record
  parent : ptr;
  child  : ptr;
  sibling : ptr;
end;
```

Type Stringptr

Variables of type string have two lengths:

- the current length that defines the number of characters in the string at any instant in time.
- the maximum length that defines the storage required for the string.

The predefined type "stringptr" defines a pointer to a string that has no maximum length associated with it until execution time. The procedure **new** is used to allocate storage for this type of pointer; an integer expression is passed to the procedure that specifies the maximum length of the allocated string.

Examples:

```
var
  p : stringptr;
  q : stringptr;
  i : 0..32767;
begin
  ..
  new(p, (i+1) div 2);
  writeln( maxlength(p@) );
    {writes '30' to output }
  new(q,5);
  q@ := '1234567890';
    {causes a truncation }
    {error at execution  }
end;
```

Type Identity and Assignment Compatibility

RT PC VS Pascal has strict type checking, which means that objects of one type cannot be combined in operations with objects of a different type. There are two major concepts described in the following sections: identical types and assignment-compatible types.

Identical Types

Two types, for example `t1` and `t2`, are considered identical under the following conditions:

- `t1` and `t2` are the same type
- `t1` is declared as synonymous with another type `t3` if `t2` and `t3` are identical.

Examples:

```
type_x = integer;  
type_y = integer;  
type_1 = set of char;  
type_2 = set of char;  
id_type = type_1;
```

In the examples, the types `type_x` and `type_y` are identical because they are defined as the same base standard type, `integer`. The types `type_1` and `type_2` are not identical because they are not simple types and occur in different type definitions. However, types `type_1` and `id_type` are identical, because `id_type` is defined as the same as `type_1`.

Assignment-Compatible Types

A value of type t_1 is considered assignment compatible with a variable of type t_2 if any of the following conditions are true:

- t_1 and t_2 are identical and do not contain a file as a component
- t_1 is a subrange of t_2
- t_2 is a subrange of t_1
- t_1 and t_2 are subranges of identical types
- t_1 is assignment compatible with integer and t_2 is real or shortreal
- t_1 and t_2 are both variable string types
- t_1 and t_2 are sets of elements of types t_3 and t_4 , and t_3 is assignment compatible with t_4 .

Defining Constants

A "literal constant" contains the actual value of the given data type. Both the integer 1776 and the string "Independence" are literal constants. A constant definition defines an identifier that is a synonym for that constant.

Using the identifier is the same as using the associated literal constant. If the string "3.14159" is a literal constant, an identifier called "pi" could be defined as a synonym for the number. The identifier is then known as a constant identifier or just a constant.

A constant may be defined by using one of the following:

- an unsigned number
- a signed number
- a constant identifier
- a signed constant identifier

a nil
a string

A constant definition occurs when an identifier is equated with a constant or a constant expression.

Examples:

```
liters_per_bottle = 0.750;    { real constant }
bottles_per_case = 12;       { integer constant }
first_vowel = 'a';           { a char constant }
cheese = 'swiss';             { a string constant }
```

A constant expression is an expression that can be evaluated by the compiler and replaced with a result at compile time. Constant expressions cannot contain a reference to a variable or to a user-defined function. They may appear in constant declarations. Constant expressions are evaluated and replaced by a single result at compile time.

Examples:

<u>constant expression</u>	<u>type</u>
ord('a')	integer
succ(chr('f0'x))	char
256 div 2	integer
'token' str(ch(0))	string
'8000'x	integer
['0'..'9']	set of char
32768*2-1	integer

Predefined Constants

RT PC VS Pascal provides constants that are automatically declared as part of the language. These constants are:

true represents the true value for a boolean type.

false represents the false value for a boolean type.

minreal	is a real constant whose value is the smallest floating-point number that can be represented on the machine.
maxreal	is a real constant whose value is the largest floating-point number that can be represented on the machine.
minint	is an integer whose smallest integer value is -2147483648.
maxint	is an integer whose largest integer value is 2147483647.

Hexadecimal Constants

Integer hexadecimal constants are enclosed in quotes and suffixed with an "X" or "x". Hexadecimal constants may be used in any context where an integer constant is appropriate. If eight hexadecimal digits (4 bytes) are not specified, RT PC VS Pascal adds the necessary number of zeros on the left. For example, 'F'x is the same as '0000000F'x and is the value 15.

Floating-point hexadecimal constants are enclosed in quotes and suffixed with an "XR" or "xr". These constants may be used in any context where a real constant is appropriate. If 16 hexadecimal digits (8 bytes) are not specified, RT PC VS Pascal adds the necessary number of zeros on the right. For example, '4110'xr is the same as '4110000000000000'xr.

String hexadecimal constants are enclosed in quotes and suffixed with an "XC" or "xc". These constants may be used in any context where a string constant is appropriate. There is one character per byte which must be specified explicitly. This takes 2 hexadecimal bits.

Binary Constants

Integer binary constants are enclosed in quotes and suffixed with a "B" or "b". A binary digit is either a '0' or a '1'. For example, 'B'x is the same as '1111'B; both represent the decimal value 15.

Structured Constants

"Structured constants" are arrays or records which contain structured fields. The type of constant is determined by the type of identifier that is used in its definition. Structured constants can be used in executable statements, constant declarations, and value declarations. There are two kinds of structured constants:

- | | |
|------------------|---|
| array constants | are specified by a list of constant expressions. Each expression defines one element of the array. |
| record constants | are specified by a list of constant expressions. Each expression defines one field of the record in the order declared. |

An element of the array can be omitted from the array or from the end of the list of constant expressions. The values of the omitted elements are not defined. To repeat the first constant expression, follow the constant with a colon and a repetition expression. The repetition expression must be a positive integer. To omit a field of the record within the list, use two commas: the value of that field is not defined.

Values within the list may correspond to fields of a record's variant part. The tag field value must be specified prior to assigning values to the variant fields. If the structured constant is imbedded within another structured constant, the type identifier can be omitted.

Examples of Structured Constants:

```
i:=59;
type
  complex = record
    re,im: real
  end;
  vector  = array[1..7] of integer;
  carray  = array[0..9] of complex;
  tetra   = array[1..3,1..2,1..4]
            of integer;
```

```

const
{ Structured Constants      }
threefour = complex(3.0,4.0);
vector_1 = vector(7,0:5,1);
vector_1 = vector(2,3,,4);
zerotetra =
    tetra(
        ( 0:4):2 ),
        ( 0:4),(0:4) ),
        ( 0,0,0,0),(0,0,0,0) ) );

{the following two declarations are equivalent}
vector_3 = carray(
    complex(1.0,0.0),
    complex(1.0,1.0):8,
    complex(0.0,1.0));
vector_4 = carray(
    (1.0,0.0),
    (1.0,1.0):8,
    (0.0,1.0));

```

Examples of Structured Constants with Variant Record Fields:

```

type
form = (fchar,finteger,freal,
        fstring);
konst =
record
size: integer;
case f: form of
fchar:      (c: char);
finteger:   (i: integer);
freal:      (r: real);
fstring:    (
    case boolean of
    true: (
        len: packed 0..32767;
        A  : alpha);
    false: string(16));
end

```

```
const
  a      = konst(1,fchar,'a');
  pi     = konst(8,freal,3.14159);
  blank  =
    konst(1,fstring,false,' ');
  stars  =
    konst(4,fstring,true,4,'****');
```


Chapter 3. Variables

This chapter describes the procedures for declaring Pascal variables in terms of the data types described in Chapter 2, "Data Types." It also describes the procedures for accessing and referencing variables of different types.

Declaring Variables

A variable has a type and a storage area in memory. At any given time, a variable takes on one value out of the collection of values that define its type. Initially, a variable has no value. It remains without a value until it is initialized by an explicit assignment.

All variables in a Pascal program must be declared explicitly prior to their use. If they are not, unpredictable results may occur. Variable declarations consist of a list of identifiers that represent the variables. This identifier is followed by the type of the variable.

variable-declaration = identifier {,identifier}: data-type;

Examples:

```
impedance : complexnumber;  
chainhead : twoway;  
treetop : symtree;  
first, last : integer;
```

```
valuefile : numbers;  
curchar : char;  
omega : real;
```

Predeclared Variables

RT PC VS Pascal contains the following predeclared variables:

- input
- output
- stderr

These are default files associated with the standard input, the standard output, and the standard error output, respectively. By default, stderr is directed to the same place as output.

These predeclared variables are covered in detail in Chapter 7, "Program Structure."

Establishing Variables

Establishing a variable involves:

1. Determining the variable's type.
2. Allocating storage for the values that the variable assumes.

"Explicitly declared variables" are automatically established on each entry to the procedure or function block in which they are declared. Global variables (those declared in the outermost block) are established only once.

Formal parameters of procedures or functions are automatically established with each invocation of that procedure or function.

Dynamic variables are explicitly established by storage management operations for type determination and storage allocation, and by assignment operations for initialization.

Lifetime of Variables

This section describes the lifetime of these variables and parameters:

- local variables
- global variables
- formal parameters
- dynamic variables
- static variables

Local Variables

The lifetime of a local variable is the same as the lifetime of the block in which it is declared. Allocation occurs on each entry to that block and deallocation occurs on each exit from that block.

Global Variables

Global variables are those variables declared in the outermost block (the program block). The lifetime of global variables is the lifetime of the entire program. Variables declared outside of procedures or functions in a segment module are considered global.

Formal Parameters

The lifetime of a formal parameter is the same as the lifetime of the procedure or function that contains the formal parameter. The formal parameter becomes established with each entry to the procedure or function and becomes undefined upon exit from the procedure or function.

Dynamic Variables

Dynamic variables are established, but not initialized, by an explicit allocation operation such as **new**. Dynamic variables become undefined when they are explicitly de-allocated by the **dispose** or **release** procedures or when no pointer variable points to them. Note that generally a pointer value has a finite lifetime that may be different from that of the pointer variable that can point to it.

Static Variables

Static variables are allocated prior to program execution and exist for the life of the program's execution. Data in static variables that are local to a routine are preserved over separate invocations of the routine. A static variable may be initialized at compile-time by the use of value declarations. See Chapter 7, "Program Structure" for a description of value declarations.

Referencing or Accessing Variables

The method by which a variable or a component of a variable is accessed differs depending on the structuring method used in the type definition for that variable. There are three basic access methods which are described in detail in subsequent text:

- An "entire variable" is a variable of a simple type (no structure). An entire variable is referenced simply by giving its name.
- A "component variable" is a variable of array, record, or file type.
- A "referenced variable" is accessed through a pointer.

Entire Variables

An entire variable is denoted by its identifier. Since an entire variable has no structure, its identifier alone is enough to reference it.

Component Variables

A component of a variable is designated by the variable itself followed by some selector that specifies the component. The form of the selector depends on the structuring method used to access the variable.

A component variable may be designated by the following selectors:

indexed-variable
field-designator
file-buffer

Referencing Indexed Variables

A component of an n -dimensional array variable is denoted by the variable followed by n index expressions. An entire array or a component of an array can be referenced by providing $n-1$ index expressions.

This occurs when an entire array or an entire subarray is passed as an actual parameter to a procedure or function.

An indexed-variable may be defined as:

indexed-variable = *array-variable subscript-list*

array-variable
is the variable name assigned to the array.

A *subscript-list* may be defined as:

subscript-list = [*expression* { *expression* }]
or
subscript-list = [*expression*] [[*expression*]]

The { *expression* } in the definition implies that there are as many expressions in the *subscript-list* as there are dimensions in the array variable. Just as in defining an array type, there are two ways to reference an array variable: with a list of subscripts separated by commas inside the brackets or with a list of bracketed subscript expressions.

The index expression types must correspond to the index types declared in the array type definition.

Examples:

ladder[top]

stairs[flight][step]

footing[left, center, right]

Referencing Strings

String variables can be referenced as single entities when the entire string is being operated upon, or single characters from a string can be referenced like a packed array of char. Values can be assigned to string variables using assignment statements, string intrinsics, the **read** or **readln** procedure, or the **get** procedure. String indexing starts from 1 so that the expression on the string *s* correctly yields the last character in the string as shown.

s[length(*s*)]

A length must never be used with a value greater than the maximum declared for that string. It is an error to reference a string *s* with an index greater than length(*s*).

It is necessary to ensure that the string length is up-to-date when a value is assigned to elements in a string beyond the string's current dynamic length. See the *RT PC VS Pascal User's Guide* for an explanation of how to modify the dynamic length of strings. The following sequence results in the null string being written instead of "x" as intended because the length of *s* is never updated to reflect the new length.

```
s := ""; s[1] = 'x'; write(s);
```

Referencing Fields of Records

A component of a record variable is denoted by the record variable followed by the component's field identifier. The field identifiers are separated by periods.

A field designator is defined as:

record-variable.field-identifier

It is an error (which is not flagged by RT PC VS Pascal) to reference a field of a variant record that is inconsistent with the tag field for that variant.

Examples:

```
{ This is a simple field reference }
```

```
impedance.realpart
```

```
{ This illustrates a reference to a field of an array of records }
```

```
bottles[milktype].dairy
```

```
{ This illustrates a deeply nested field reference }
```

```
king_caractacus.court.ladies.faces.noses
```

Referencing File Buffers

At any time, only the component determined by the current file position is directly accessible. This component is known as the "current file component" and is represented by the file's buffer variable.

A file-buffer is defined as:

file-buffer = *file-variable* @
or
file-buffer = *file-variable* ->

file-variable
specifies a variable.

Pointer Referenced Variables

A referenced-variable is defined as:

referenced-variable = *pointer-variable* @
or
referenced-variable = *pointer-variable* ->

pointer-variable
specifies a variable.

If p is a variable that is a pointer to type t , p indicates the pointer variable and its pointer value. However, $p@$ indicates the variable of type t that p references.

Examples:

{ Left node in the tree variable }

treetop.leftnode@

{ Gets long side of blackboard }

cue@.longside

Chapter 4. Expressions

An "expression" is a construct that defines the rules of computation for creating a value by performing operations on operands. (Operations are specified by operators, and operands are specified by variables, constants, and function references.) These newly-created values can then be used in assignment statements or in conditional expressions to control subsequent program actions.

These expressions may be used in RT PC VS Pascal.

- An unsigned constant which may be one of the following:

unsigned number
string
constant identifier
nil

- A factor which is one of the following:

variable
unsigned-constant
function-designator
set-constructor
(expression)
not factor

- A set constructor which is the following:

[*element* {,*element*}]

- An element which is one of the following:

expression
expression..expression

- A term which is one of the following:

factor
term multiplication-operator factor

- A simple-expr which is one of the following:

term
simple-expr addition-operator term
addition-operator term

- An expression which is one of the following:

simple-expr
simple-expr relational-operator simple-expr

Operators in Expressions

Operators perform operations on a value or a pair of values to produce a new value. Most operators are defined only on basic types, though some are defined on more types. An operation on a variable or field that has an undefined value produces an undefined result.

The following sections define the applicable range as well as the results of the defined operators.

Order of Evaluation in Expressions

The rules of composition for expressions specify operator precedence according to five operator classes. The following lists the operator precedence from the highest to the lowest.

- the not operator — highest
- the unary operators
- the multiplication operators

- the addition operators
- the relational operators — lowest.

Operators at the same precedence level are applied left to right, except where parentheses are used to override the normal order of evaluation.

The precise order of operand evaluation is undefined. Some operands may not be evaluated at all, which occurs when the value of the expression can be determined without the value of that particular operand.

Not Operator

The **not** operator applies to factors of type boolean, integer, or set. When applied to type boolean, the meaning is negation; that is, `not true = false` and `not false = true`. When applied to type integer, the **not** operator negates all the bits in the value; that is, it performs a ones-complement negation of each bit in the operand. When applied to type set, the meaning is a complement of the set; that is, if set `x` is `[1, 2, 3, 4]` and it is currently `[1]`, the complement of set `x` is `[2, 3, 4]`. The result of applying the **not** operator to a value of type integer is type integer as shown in Figure 4-1.

Operator	Operation	Operands	Result
<code>~ (not)</code>	boolean not	boolean	boolean
<code>~ (not)</code>	logical one's complement	integer	integer
<code>~ (not)</code>	set complement	set of <code>t</code>	set of <code>t</code>

Figure 4-1. NOT Operator

Unary Operators

The + (plus) and - (minus) signs can be used as unary operators. Figure 4-2 shows the unary operators, their permissible operand types, and their result types.

Operator	Operation	Operands	Result
+	identity	real, shortreal , or integer	real, shortreal , or integer
-	negation	real, shortreal , or integer	real, shortreal , or integer

Figure 4-2. Unary Operator

These operators apply to integer, real, and **shortreal** data types only. Applying a unary operator to a data type produces a result that is the same data type as that of the operand.

Multiplication Operators

The multiplication operators have the next highest precedence after the unary operators. Figure 4-3 shows the multiplication operators, the permissible types of their operands, and the result types.

Operator	Operation	Operands	Result
*	multiplication	real, shortreal , or integer	real, shortreal , or integer
	set intersection	any set type t	t

Figure 4-3 (Part 1 of 2). Multiplication Operator

Operator	Operation	Operands	Result
/	division	real, shortreal or integer	real, shortreal
div	division with truncation	integer	integer
mod	modulus	integer	integer
and	logical and bitwise and	boolean integer	boolean integer
& (and)	boolean and logical and	boolean integer	boolean integer
	string catenation	string	string
<<	logical left shift	integer	integer
>>	logical right shift	integer	integer

Figure 4-3 (Part 2 of 2). Multiplication Operator

Operands of the * (multiplication) and / (division) operators can be mixed integer, real, and shortreal data types. If both operands of the * operator are type integer, the result is type integer. If either operand is type real, the other operand is converted to type real and the result is type real. If both operands are of type shortreal or one is of type shortreal and the other of type integer, the result is of type shortreal.

The result of the / operator is either real or (in the case when one or both operands are of type shortreal) shortreal.

The **div** operator applies to values of type integer only and represents truncating division; **div** always truncates towards zero. It is an error to divide by zero. If the signs of the operands are the same, the result is positive; if the signs are different, the result is negative.

The **mod** operator defines the modulus operation between two values of type integer. It is an error if the right operand of **mod** is zero. The interpretation of **mod** is:

$$a \bmod b = a - (a \operatorname{div} b) * b$$

When applied to operands of type boolean, the **and** operator produces a result of type boolean. When applied to operands of type integer, however, the **and** operator performs a bitwise-logical and on the operands and produces a result of type integer.

Addition Operators

The addition operators have the next highest precedence after the multiplication operators. Figure 4-4 shows the addition operators and their permissible operand types.

Operator	Operation	Operands	Result
+	addition	real, shortreal , or integer	real, shortreal , or integer
	set union	any set type t	t
-	subtraction	real, shortreal , or integer	real, shortreal , or integer
	set difference	any set type t	t
or	logical or bitwise or	boolean integer	boolean integer
&& (xor)	exclusive or 'exclusive' union	integer set of t	integer set of t
(or)	boolean or logical or	boolean integer	boolean integer

Figure 4-4. Addition Operator

Operands of the + (addition) and - (subtraction) operators can be mixed integer, real, and **shortreal** data types. If both operands of the + or - operator are of type integer, the result is of type integer. If either operand is of type **shortreal**, the other operand is converted to type **shortreal** and the

result is also of type **shortreal**. Otherwise, if either operand is of type real, the result is also of type real.

When applied to operands of type boolean, the **or** and **xor** operators produce a result of type boolean. When applied to operands of type integer, however, the **or** operator performs a bitwise-logical or on the operands and produces a result of type integer.

Relational Operators

Figure 4-5 shows the relational operators, their permissible operand types, and the result type.

Operator	Operation	Operands	Result
=	compare equal	any set, scalar type, pointer or string	boolean
< > (~=)	compare not equal	any set, scalar type, pointer, or string	boolean
<	compare less than	scalar type, string	boolean
< =	compare < or = subset	scalar type, string, 'set of' t	boolean
>	compare greater	scalar type, string	boolean

Figure 4-5 (Part 1 of 2). Relational Operator

Operator	Operation	Operands	Result
> =	compare > or = superset	scalar type, string, 'set of' t	boolean
in	set membership	t and 'set of' t	boolean

Figure 4-5 (Part 2 of 2). Relational Operator

Note that all scalar types define ordered sets of values.

Comparison of Scalars

All six relational operators <, <=, >, >=, =, and <> are defined between operands of the same scalar type.

For operands of type integer, real, or **shortreal**, the operators have their usual meaning. Operands of integer and real data types are considered to form a hierarchy, with the integer data type at the bottom, the **shortreal** data type in the middle, and the real data type at the top.

If the operands are of different numeric types, the lower type of operand is converted to the level of the other operand type before the comparison. For example, in this expression

```
shortreal type < real type
```

the **shortreal** operand is converted to real before the comparison is made.

For operands of type boolean, the relation `false < true` defines the ordering.

For operands of type char, the relation `a op b` holds if, and only if, the relation `ord(a) op ord(b)` holds when `op` denotes any of the six comparison operators and `ord` is the mapping function from type char to type integer defined by the ASCII collating sequence.

For operands of any ordinal type "t", `a = b` if `a` and `b` are the same value; and `a < b` if `a` precedes `b` in the ordered list of values that define "t".

Comparison of Booleans

If p and q are boolean expressions, $p = q$ means equivalence, and $p \leq q$ means implication of q by p . If p is true, the value of q becomes true. If p is false, the value of q remains unchanged.

Direct Pointer Comparison

Two direct pointers can be compared if they are pointers to identical types. To compare pointers of differing types, find their **ord** value as described in Chapter 8, "Built-In Procedures and Functions." Pointers may be compared for equality or inequality only. Two pointers with the value nil are always equal.

String Comparison

All six relational operators may be applied to string operands. The relational operators compare both packed array of char and string values.

In the case of a packed array of char, both operands must be the same size. The maximum length of string comparison of values of packed array of char is 32767 characters. For example, this variable whose declaration is:

```
var  
strtype: packed array [-1..32767] of char;
```

specifies the largest string variable that can be compared in one operation.

In the case of string comparison, the operands may be of different lengths. If the operands are of different lengths, trailing spaces are significant. For example, the string

```
'a'
```

compares less than the string

```
'a '
```

Comparison of string operands or packed array of char operands denotes alphabetical ordering according to the ASCII character set collating sequence.

Because a **string data type** is represented differently from a packed array of char, they cannot be compared with each other. On the other hand, a character string constant is of ambiguous type. Therefore, a string constant can be compared either to a string operand or to a packed array of char operand because the type of the string constant is converted to the type of the other operand in the comparison operation.

Set Comparison

The relation `scalar_value in some_set` is true if the `scalar_value` is a member of `some_set`. The base type of the set must be the same as the base type of the scalar.

The set operations are defined between two set values of the same base type. For two sets (`s1` and `s2`) of the same base type, they are:

- | | |
|-----------------------------|--|
| <code>s1 = s2</code> | is true if all members of <code>s1</code> are contained in <code>s2</code> and all members of <code>s2</code> are contained in <code>s1</code> . |
| <code>s1 <> s2</code> | is true when <code>s1 = s2</code> is false. |
| <code>s1 <= s2</code> | is true if all members of <code>s1</code> are also members of <code>s2</code> . |
| <code>s1 >= s2</code> | is true if all members of <code>s2</code> are also members of <code>s1</code> . |
| <code>s1 in s2</code> | is true if <code>s1</code> is a member in <code>s2</code> . |

Non-Comparable Types

The following RT PC VS Pascal types cannot be compared.

- files
- arrays
- variant records
- records containing fields of non-comparable types
- records containing fields in different order.

Note: The exception to this rule is that packed array of char operands can be compared if they are the same size.

Out-of-Range Values

It is possible that expression evaluation can yield results which are outside of the range of values for a given data type. Expressions involving the real and double data types can generate several different extreme values. The extreme value of +infinity or -infinity is a result of either overflow or dividing a nonzero value by 0.0.

Compile-Time Constant Expressions

The RT PC VS Pascal compiler evaluates certain types of integer and boolean constant expressions at compile time. Integer expressions consisting of constant expression operands and the following operators are folded into constant expressions:

Binary Operators = <> + - * /

Unary Operators - +

Boolean expressions consisting of constant expression operands and the following operators are folded into constant expressions:

Binary Operators = <> and or

Unary Operators ~

Dead Code Elimination

The RT PC VS Pascal compiler recognizes the following code:

```
if false then
  statement_1
else
  statement_2
```

and generates code for `statement_2` only. Similarly, if the boolean expression is true, only `statement_1` is generated. Constant expressions that fold into constants are recognized as constant true or false. This feature facilitates keeping several versions of similar source code in the same file without adding extra generated code when the code is compiled.

Examples:

```
const
  version = 10;
.
.
if version = 7 then
  writeln('Too old!')
else
  writeln('Not too old!');
```

This code fragment, with the constant "version" set equal to 10, has the same effect as the following code fragment:

```
writeln('Not too old!');
```

Chapter 5. Statements

Each RT PC VS Pascal program must include at least one statement. Statements denote algorithmic operations and define the actions to be performed on program objects that have been introduced by type and variable declarations.

This chapter describes statement labels, assignment statements, procedure reference statements, structured statements, and control statements.

Statement Labels

A statement can be labeled by preceding it with an unsigned integer constant in the range 0..9999 or an identifier followed by a colon. The statement can then be explicitly referred to by a **goto** statement. When executed, this `goto label` sequence causes the statement prefixed by the label to be executed as the next statement in the program instead of the statement following the **goto**. In other words, the labeled statement is **goto**'s successor.

Scope Of Statement Labels

The scope of a statement label is the body of the procedure or function in which the label is declared and all of its nested procedures and functions. This means that a **goto** statement cannot transfer control into a procedure or function unless that procedure or function has been activated.

Assignment Statements

The assignment statement replaces the current value of a variable with a new value derived from an expression evaluation, or it defines the value that a function variable returns.

An assignment statement is defined as:

variable := expression
or
function-identifier := expression

Assignments to Variables and Functions

The part to the left of the assignment symbol (*:=*) is evaluated to obtain a reference to some variable. The expression on the right side is evaluated to obtain a value. The referenced variable's current value is discarded and replaced with the expression's value.

The variable on the left side of an assignment statement must be assignment compatible, described in Chapter 2, "Data Types," with the type of the expression on the right side.

A string constant may be assigned to a variable of type `packed array [1..n] of char`, provided that the string value is the same length as the array object. The maximum length of such an assignment is 32767 characters.

Examples:

x := 5;	{ assignment to variable }
y := x * 10 + 18;	{ assignment of expression }
ch := chr(10);	{ assignment of function value }
rope := 'hemp';	{ string assignment }
poke@ := nil;	{ destroy the system vector }

Procedure Reference Statements

A procedure reference statement creates an environment for execution of the specified procedure and transfers control to that procedure.

A procedure reference or procedure call statement is defined as:

procedure-identifier[*actual-parameter-list*]
or
procedure-identifier

actual-parameter-list

specifies the actual parameters to be used in the procedure. An actual parameter may be one of the following:

expression
procedure-identifier
function-identifier

The *actual-parameter-list* must be compatible with the formal parameter list of the procedure. An actual parameter corresponds to the formal parameter that occupies the same ordinal position in the formal parameter list.

Only formal parameters that are value parameters can have an actual parameter that is an *expression*. Value parameters must be assignment compatible with the type of the formal parameter.

Formal parameters that are **var** parameters must have actual parameters that are identical types. In addition, the actual parameters cannot be components of packed objects.

Note: Components of packed objects can be passed as **var** parameters in IBM mode only.

Structured Statements

Structured statements are constructs composed of statement lists. They provide scope control, selective execution, or repetitive execution of the constituent statement lists.

RT PC VS Pascal contains several types of structured statements which are explained in detail in this section. These structured statements are:

- begin-end statements
- case statements
- empty statements
- for-do statements
- if-then-else statements
- repeat-until statements
- while-do statements
- with statements

Begin-End — Compound Statements

A **begin** statement specifies execution of a statement list. The reserved words **begin** and **end** mark the start and finish of the statement list. Exiting from the statement list is done either by completing execution of the last statement in the statement list or by explicitly transferring control.

A **begin** statement is defined as:

begin *statement-list* **end**

statement-list

is a statement or a series of statements separated by semicolons.

Case Statements

A **case** statement selects one of its component statements for execution depending on the value of an expression. The expression is known as the case selector. Each of the component statements is tagged with one or more simple scalar constants. The tags are known as selection specifications (selection specs).

If the value of the selector matches that of one of the statement tags, that statement is executed. If the selector value does not match any of the statement selection specifications, **the statement (if any) following an otherwise symbol is executed.**

A **case** statement is defined as:

case *expression of cases*
 [**otherwise:** *statement*] **end**

expression

is known as the selector and must yield a value or an ordinal type.

cases

are the case labels and must be of the same unstructured type as the selector. A case is defined as:

```
selection-spec {, selection-spec} :  
    statement;  
or  
selection-spec..selection-spec :  
    statement;
```

selection-spec

is used to tag the component statements. This is not a label in the RT PC VS Pascal sense, and, as such, cannot be used as the target of a **goto** statement. Also, it should not appear in any label declaration part. A *selection-spec* must be a scalar constant.

selection-spec..selection-spec

specifies a range of values to be tagged to the component statement as shown in the following example.

```
case i of  
  1..5:   choice1;  
  6..10:  anything;  
end;
```

statement

is used to evaluate the *expression* and to execute the statement labeled by the resulting value.

Case selectors and the statement tags must be non-real scalar types. In addition, the case selectors and the statement tags must be of assignment-compatible types.

Examples:

```
case wine_type of
    champagne:
        anything_goes;

    cabernet:
        roast_lamb;

    chardonnay:
        veal_piccata;

otherwise:
    hamburger;
end;
```

Empty Statements

An **empty** statement is an extra statement used by itself. It is used as a place holder and has no effect on the execution of the program. This statement can be used to place a label in a program, but the label is not attached to another statement (such as, at the end of a compound statement). It avoids the ambiguity that arises in nested **if** statements. The **empty** statement may also be used to force an **else** clause to be paired with an outer nested **if** statement by using an **empty** statement after an **else** in the inner nested **if**.

Example:

```
if b1 then
    if b2 then
        s1
    else;
else
    s2
```


For-Do Statements

The **for** statement executes its subordinate statement repeatedly while a progression of values is assigned to a control variable of the **for** statement.

A **for** statement is defined as:

for *control-variable* := *for-list* **do** *statement*

control-variable

is an identifier which is set to the initial value. After every iteration, the *control-variable* is either incremented or decremented until its value is greater than or less than the final value.

for-list

contains the *initial* and *final* values. These values can specify either an ascending or descending succession.

initial-value

final-value

are expressions which yield a value of the same ordinal type as the *control-variable*.

The *control-variable*, the *initial-value*, and the *final-value* must all be of the same scalar type or a subrange of that scalar type. No part of the statement controlled by the **for** statement may alter the *control-variable* during the execution of the **for** statement.

Neither the *control-variable*, the *initial-value*, nor the *final-value* may be of type real. The *control-variable* must be local to the procedure or function that contains the **for** statement.

The value of the *control-variable* is undefined on normal termination from the **for** statement. If the **for** statement is exited prematurely (by a **goto** statement), the value of the *control-variable* is defined.

Examples:

```
{ Initialize an array to zero }  
  
for index := 1 to 100 do  
    row[index] := 0  
  
  
{ Scan from the end of an array }  
  
for where := 200 downto 1 do  
    if what[where] = thing then  
        foundit := true
```

If-Then-Else Statements

The **if** statement specifies that another statement is or is not to be executed depending on the truth value of a conditional expression. If the value of the conditional expression is true, the statement is executed. If the value of the conditional expression is false, either subsequent statements are not executed or the statement following an **else** clause is executed.

The **if** statement is defined by:

```
if boolean-expression then statement  
or  
if boolean-expression then statement  
    else statement
```

Because RT PC VS Pascal statements are open forms, it is possible to construct a chain of **else if** clauses to select "one out of many different conditions".

In common with other languages, RT PC VS Pascal has the "dangling else" situation. This situation arises when one **if** statement contains a subordinate **if** statement and there is only one **else** clause. In RT PC VS Pascal, the **else** clause matches the most recent **if** statement that does not have an **else** clause. The following examples illustrates this.

Examples:

```
{ A simple if statement }

if day in [Monday..Friday] then
  get_up_and_go
else
  roll_over

{ An if statement with a compound block }

if sun > yardarm then
  begin
    brew_coffee;
    prepare_snacks;
    relax;
  end
else
  sleep_on

{ An else if chain }

if weather = raining then
  sleep_in
else if lawn = wet then
  clip_the_hedge
  else if grass > 6 then
    mow_the_lawn
  else
    turn_on_lawn_sprinklers

{ A dangling else clause }

if condition_1 then { 1 }
  if condition_2 then { 2 }
    if condition_3 then { 3 }
      ...statements...
    else { Goes with statement 3 }
      ...statements...
  else { Goes with statement 2 }
    ...statements...
else { Goes with statement 1 }
  ...statements...
```


Repeat-Until Statements

The **repeat** statement controls the repetitive execution of a list of statements. The statements are executed until the condition at the end of the statement evaluates to true.

The **repeat** statement is defined as:

repeat *statement-list* **until** *expression*

The *expression* controlling repetition must be of type boolean. The *statement-list* between the **repeat** and **until** symbols is executed repeatedly until the *expression* becomes true.

Note that the body of a **repeat** statement is always executed at least once since the termination test is at the end. Contrast this behavior with the **while** statement described in “While-Do Statements.”

Example:

```
repeat
  consume_platefull;
  refill_plate;
until (food_amount <= 0) or (me = full);
```

While-Do Statements

A **while** statement controls repetitive execution of another statement until evaluation of a boolean expression becomes false.

A **while** statement is defined as:

while *expression* **do** *statement*

The *statement* is repeated while the value of *expression* remains true. The *expression* must be of type boolean. When *expression* becomes false, control passes to the statement after the while statement.

If the value of *expression* is false at the time that the **while** statement is encountered for the first time, the subordinate statement is never executed. Thus the **while** statement provides a means to "do nothing gracefully". Contrast this behavior with the **repeat** statement described in "Repeat-Until Statements."

Example:

```
while bytes_to_go > 0 do
begin
  if bytes_to_go <= blocksize then
    transferlength := bytes_to_go
  else
    transferlength := blocksize;
  dotransfer;
  bytes_to_go := bytes_to_go - transferlength;
  blocknumber := blocknumber + 1
end;
```

With Statements

The **with** statement provides a "shorthand" notation for referring to fields in a record. The effect of the **with** statement is to insert implicitly the required record identifier before the name of each field. The **with** statement effectively opens the scope that contains the corresponding field identifiers for each specified record variable.

The **with** statement is defined as:

```
with record-variable {,record-variable}
do statement
```

record-variable
is the record identifier.

Within the body of the **with** statement, fields of the specified *record-variable* do not need to be qualified by the name of the record.

If there is a local variable "x" and a field "x" in a record "r" that is the subject of a **with** statement, the statement:

```
with r do
```

disregards the local variable "x" until the end of the **with** statement.

A **with** statement that has multiple record variable fields is interpreted as nested **with** statements. The statement:

```
with record_1, record_2, record_3 do
```

is equivalent to the statement:

```
with record_1 do
  with record_2 do
    with record_3 do
      ...statement...
```

Example:

```
var
  treetop: symtree;

  with treetop do
    begin
      leftnode := nil;
      rightnode := nil
    end { with }
```

This is a shorthand for the following statements:

```
treetop.leftnode := nil;
treetop.rightnode := nil;
```


Control Statements

RT PC VS Pascal supports the following control statements which are explained in detail in the following text.

- `assert`
- `continue`
- `goto`
- `leave`

Assert Statements

The **assert** statement is used to check for a specific condition and signal a run-time error if the condition is not met. The condition is specified by the expression which must evaluate to a boolean value. If the condition is not true then the error is raised. The compiler may remove the statement from the program if it can be determined that the assertion is always true.

Example:

```
assert a >= b
```

Continue Statements

The **continue** statement causes a jump to the loop-continuation portion of the inner-most enclosing **while**, **for**, or **repeat** statement. In other words, it is a **goto** to the end of the loop. The following examples illustrate the **continue** statement function in each of the loop constructs.

Examples:

```
{ Continue used in a while loop }
  while expr do begin
    ...
    continue
    ...
  { continue jumps to here }
end;

{ Continue used in a for loop }
  for i := expr1 to expr2 do
  begin
    ...
    continue
    ...
  { continue jumps to here }
end;

{ Continue used in a repeat loop }
  repeat
    ...
    continue
    ...
  { continue jumps to here }
until expr;
```

Goto Statements

The **goto** statement names a labeled statement as its successor.

The **goto** statement is defined as:

goto *label*

label

is an unsigned integer in the range 0 to 9999 or an identifier.

Note: The scope of a *label* is the procedure in which that *label* is defined

and all its nested procedures and functions. Therefore jumping into a procedure is not allowed if that procedure is not activated.

Every *label* in a procedure must be declared in the label declaration part at the head of the procedure.

Example:

```
if status = error then
  goto 9999  { Exit to end of procedure }
```

Leave Statements

The **leave** statement causes an immediate, unconditional exit from the innermost enclosing **for**, **while** or **repeat** loop. The following code segments are equivalent:

This code

```
while expr do
  begin
    ...
    leave
  end;
```

is equivalent to this code:

```
while expr do
  begin
    ...
    goto lab;
  end;
lab;;
```


Examples:

```
p:=first;
while p<>nil do
  if p@.name = 'Joe Smith' then
    leave
  else
    p:=p@.next;
{ p either points to the desired }
{ data or is nil }
```


Chapter 6. Input and Output

An RT PC VS Pascal program communicates with the world outside the computer system through input and output facilities — "reading" and "writing".

In addition to supporting the input/output facilities as defined by standard Pascal, RT PC VS Pascal supports untyped (block access) files and random access to typed files.

The File Buffer Variable

An RT PC VS Pascal file of "some_type" is a sequential file: its components appear in strict sequential order. (The **seek** procedure is ignored for the duration of this discussion). Writing implies appending a component to the end of the file. Reading implies that the next component in sequence is obtained from the file. The following discussion applies only to typed files.

Associated with each typed file variable is an implicit "buffer variable", also known as the file "window". The buffer variable can be thought of as a placeholder where the current file component is held. The buffer variable holds the next available component when reading. When writing, it holds the component that is to be appended to the file by a **put** procedure call.

For a given file variable *f*, the buffer variable is referenced by the notation *f@* or *f->*. In the following example when the file `phone` is opened for reading by the **reset** procedure call, the first component of the file is in the buffer variable.


```

type
  people = file of names;

var
  phone    : people;
  curcomp  : names;

reset (phone);

```

An assignment statement of the form:

```
curcomp := phone@;
```

assigns the contents of the buffer variable to the variable `curcomp`. The contents of the buffer variable then become undefined. The next component from the file is moved into the buffer variable by a `get` procedure call.

When the file `phone` is opened for writing by the `rewrite` procedure call, the buffer variable is undefined. An assignment of the form:

```
phone@ := curcomp;
```

assigns the value of the variable "curcomp" to the buffer variable. A subsequent `put` procedure call appends the contents of the buffer variable to the file "phone". The contents of the buffer variable become undefined until another assignment defines it.

General File Handling Routines

This section covers the RT PC VS Pascal routines for handling files of any type. The following list summarizes these routines.

close	closes a file.
eof	determines if end-of-file read.
get	gets component from a file.

put	replaces component to a file.
reset	opens existing file for input.
rewrite	opens a file for output.
update	opens a file for input and output.

Close — Closes a File

The **close** removes the association of a file variable with an external file. It marks the file as closed. The file variable for that file is then undefined. If a file is already closed, a **close** call does nothing. The file must be opened prior to using it again.

The **close** procedure is defined as:

```
procedure close(file);
```

file
is a file variable.

Eof — Determines if End-of-File Read

The **eof** (end-of-file) function returns true if a **read** or a **get** from a file encounters an end-of-file. It returns false in all other cases. To set the **eof** to true for a file attached to the console, the **eof** character must be typed. In RT PC VS Pascal this is **Control-D**. For a text file, a return of true for **eof** implies that **eoln** is true as well.

If a file is closed, **eof** returns true. After a **reset** function takes place, **eof** is false for the **reset** file. If **eof** becomes true during a **get** or a **read**, the data obtained is not valid.

The **eof** function is defined as:

```
function eof (f);  
or  
function eof ();
```

f
is a var parameter of any file type.

The **eof** function may be called without a parameter. In this case, the default file "input" is used.

Get — Gets Component from a File

The **get** procedure obtains the next element from a file, assuming there is a next element to be obtained. It advances the current file position to the next component in the file. The value of this component is then assigned to the buffer variable "file@".

If there is no next component in the file, the value of the buffer variable is undefined and the **eof** becomes true.

If the **eof** is already true, a **get** or an attempt to read past the end-of-file has an undefined result.

A **get** invocation on a file of type text returns a single character.

A **get** procedure is defined as:

```
procedure get(file);
```


Put — Replaces Component of a File

The **put** procedure replaces the value of the buffer variable *file@* in the file in the position of the last **get** from the file or in the position after the previous **put** to the file. The value of *file@* becomes undefined after the call to **put**.

A call to **put** by a file of type text transfers a single character. The file must have been previously opened for output.

The **put** procedure is defined as:

```
procedure put(file);
```

Reset — Opens an Existing File for Input

The **reset** procedure opens the *file* and positions the file pointer at the beginning of the file. If the file variable was previously opened, access to the previous file is lost and no close is done. For a proper closing of any opened file, a specific call to **close** must be done. If the file is not empty, the **eof** becomes false. If the file is empty, the **eof** becomes true.

The **reset** procedure is defined as:

```
procedure reset(file [, string]);
```

file

is the RT PC VS Pascal file variable.

string

is used to specify any special file dependent options that may be used in opening the file. The available open options are: DDNAME=, INTERACTIVE, NAME=, and UCASE. Consult the *RT PC VS*

Pascal User's Guide for information about these options and a description of how and when to use them.

Rewrite — Opens a File for Output

The **rewrite** procedure creates a new file of a specified name and discards any existing file of the same name. It does this by discarding the current value of the file variable *file* effectively creating a new file.

If the variable was previously opened, access to the previous file is lost and no close is done. For a proper closing of any opened file, a specific call to **close** must be made.

The **rewrite** procedure is defined as:

```
procedure rewrite(file [, string]);
```

file

specifies the RT PC VS Pascal file variable.

string

is used to specify any special file dependent options that may be used in opening the file. The available open options are: DDNAME=, INTERACTIVE, NAME=, and UCASE. Consult the *RT PC VS Pascal User's Guide* for information about these options and a description of how and when to use them.

Update — Opens File for Input and Output

The **update** procedure opens a designated file for both input and output updating.

The **update** procedure is defined as:

```
procedure update(file [, string]);
```

file

is a variable of type text.

string

is a string value that specifies file dependent options that may be used in opening the file. The available open options are: DDNAME=, INTERACTIVE, NAME=, and UCASE. Consult the *RT PC VS Pascal User's Guide* for information about these options and a description of how and when to use them.

The **update** procedure opens a file for both input and output (updating). A **put** operation replaces a file component obtained from a preceding **get** operation. The execution of **update** causes an implicit **get** of the first file component.

Note: Updating a file of type "text" is not allowed.

Examples:

```
var
  filevar : file of record
    cnt : integer;
    ...
begin
  ...
  update(filevar); {open and get      }
  while not eof(filevar) do
    begin
      filevar @.cnt :=filevar@.cnt+1;
      put(filevar); {update last elem}
      get(filevar); {get next elem   }
    end;
```


Text File Handling Routines

RT PC VS Pascal provides standard routines for controlling text file input and output, random access, and end-of-line. These routines apply to files of type text and are found in the following list:

cols	determines current column.
eoln	determines if end-of-line read.
page	skips to new page.
read and readln	read character strings and convert them into internal representations.
seek	allows random access to files.
termin	opens file for input.
termout	opens file for output.
write and writeln	append character strings to a text file.

Cols — Determines Current Column

The **cols** function returns the current column number (position of the text character to be written) on the output file designated by the file variable.

The **cols** function is defined as:

```
function cols(var f: text) : integer;
```

f
is a text file set to output.

The output may be directed to a specific column with the following code:

```
if tab > cols(f) then
  write(f, ' ':tab-cols(f)); { Write blank at tab minus
                              current column }
```

Note: The file variable can never be a default on the **cols** function.

Eoln — Determines if End-of-Line Read

The **eoln** function returns true if the text file position is at an end-of-line character. Otherwise the **eoln** function returns false. If **eoln** is true, the value in the file buffer is a blank. The **eoln** function is only defined for files whose components are of type text or interactive.

The end of line is determined by reading the ASCII character reserved for this function. In RT PC VS Pascal this is a **Carriage Return**. See the *RT PC VS Pascal User's Guide* for a list of all the standard ASCII characters.

The **eoln** function is defined as:

```
function eoln (f);
or
function eoln ();
```

f

is a var parameter which must be of type text or interactive.

The **eoln** function may be called without a parameter. In this case, the default file "input" is used.

Page — Skips to New Page

The **page** procedure may be used to skip to the top of a new page on a text or interactive file. A call to **page** outputs a single ASCII form feed character (0C) to the specified file. For many devices, this results in a form feed; however, not all devices recognize this character as a form feed.

The **page** procedure is defined as:

```
procedure page(file: text);  
or  
procedure page();
```

The **page** procedure may be called without a parameter. In this case, the default file "input" is used.

Read and Readln Procedures

The **read** and **readln** procedures read character strings representing numbers from a text file and convert them into their internal representations. The **readstr** procedure reads data from a string and is described in detail in the string manipulation function in Chapter 8, "Built-In Procedures and Functions." The **readln** procedure differs from the **read** procedure only in that **readln** positions the file at the beginning of the next line after completing the **read**. Multiple variables, separated by commas, may be used on each call. The effect is the same as multiple calls to **read**.

The **read** and **readln** procedures are defined as:


```

procedure read
procedure read(f : text; v : variable);

procedure readln
procedure readln(f : text; v : variable);
procedure readln(f : text );

```

f

is an optional text file that may be used as input.

Note: If this parameter is omitted, the default file "input" is used.

v

is one or more variables, separated by commas. Each variable must be one of the following types:

```

integer (or subrange)
char (or subrange)
real
shortreal
string
packed array of char

```

A read procedure call

```
read (v1, v2, ..., vn)
```

is equivalent to:

```
read (input, v1, v2, ..., vn)
```

The following **read** procedure call

```
read (file, v1, v2, ..., vn)
```

is equivalent to this sequence of **read** procedure calls:

```

read  (file, v1);
read  (file, v2);...
read  (file, vn);

```

If `ch` in the following example is a variable of type `char`, these two programs are equivalent.

Program 1

```

var
  ch: char;
  rasp: file of char;
begin
  read(rasp, ch);

  end.

```

Program 2

```

var
  ch: char;
  rasp: file of char;
begin
  ch := rasp@;
  get(rasp);

  end.

```

If `v` in the following example is a variable of type integer, a subrange of integer, shortreal, or real, the procedure reference

```
read(file, v)
```

reads a sequence of characters from the file referenced by `file`. The sequence of characters should form a valid number according to RT PC VS Pascal's rules for numbers described in Chapter 1, "Introduction."

When the number is formed, it is assigned to the variable `v`. Blank lines and spaces preceding the number are skipped in the file. Reals are read in the same way as integers. Booleans cannot be read by a `read` or `readln` call. Structured types cannot be read.

Reading Integer Data

Integer data from a text file is read by scanning off leading blanks, accepting an optional sign, and converting all characters up to the first non-numeric character or end-of-line character.

Reading Char Data

A variable of type `char` is assigned the next character in the file.

Reading String Data

Characters are read into a string variable until the variable reaches its maximum length or until the end-of-line is reached.

Reading Real (Shortreal) Data

Real (shortreal) data is read by scanning off leading blanks, accepting an optional sign, and converting all characters up to the first non-numeric character not conforming to the syntax of a real (shortreal) number.

Reading from a File of Any Type

The **read** procedure can also read from a file of any type.

A **read** procedure call of the form:

```
read (file, v1, v2, ..., vn)
```

is equivalent to the sequence:

```
v1 := file@;  
      get(file);  
v2 := file@;  
      get(file);  
...  
vn := file@;  
      get(file);
```

where the v_n contains the list of variables to read into a file.

Note: The type of each variable in the list must be identical to the type of the elements in the file.

Reading Packed Array of Char Data

When a variable is declared as a packed array [1..n] of char, characters are stored in each element of the array. This is equivalent to a loop ranging from the lower bound of the array to the upper bound which performs a **read** operation for each element. If the end-of-line (**eoln**) condition should become true before the variable is filled, the rest of the variable is filled with blanks.

Examples:

```
var
  i,h: integer;
  s: string(100);
  ch: char;
  cc: packed array [1..10] of char;
  f: text;
```

```
  .
  .
  readln(f,i,j,ch,cc,s);
```

(* Assuming the data is: 36 24 abcdefghijklmnopqrstuvwxyz
the variables are assigned as follows:

i	36	
j	24	
ch	' '	
cc	'abcdefghij'	
s	'klmnopqrstuvwxyz'	
length(s)	16	*)

Reading Variables with a Length

A variable of **read** may be qualified with a field length expression. This expression denotes the number of characters in the input line to be processed for that variable. If the number of characters indicated by the field length is exhausted during a **read** operation, then the reading operation stops so that a subsequent **read** begins at the first character following the field. If the **read** completes prior to processing all characters of the field, then the rest of the field is skipped.

The **read** procedure with a specified field length is defined as:

```
procedure read(f,v:n);
```

f
is the name of the file.

v
is the variable being read.

n
is the field length expression.

Examples:

```
var
  i,j: integer;
  s: string(100);
  ch: char;
  cc: packed array[1..10] of char;
  f: text;
  .
  .
  readln(f,i:4,j:10,ch:j,cc,s);
```

(* Assuming the data is: 36 24 abcdefghiklmnopqrstuvwxyz
the variables are assigned as follows:

i	36	
j	4	
ch	'i'	
cc	'nopqrstuvwxyz'	
s	'xyz'	
length(s)	3	*)

Seek — Allows Random Access to Files

RT PC VS Pascal supports random access to files of specific types. The **seek** procedure has two parameters: the file variable and an integer specifying the record number to which the file window should be moved. A **seek** can be used only with typed files that are not text files.

The **seek** procedure is defined as:

```
procedure seek(file: file-type, position: integer);
```

file

is the file variable for the specified file.

position

is the number of the record to which the file window is to be moved. Records are numbered sequentially from one.

The **seek** moves the file window to the *position*th record in the file specified by *file*. The **eof** and **eoln** are set to false.

An attempt to **put** a record beyond the physical end-of-file sets the **eof** true.

If a **get** or **put** is not performed between two **seek** procedures, the contents of the file window are undefined.

Termin — Opens File for Input

The **termin** procedure opens a designated file for interactive input from the terminal.

The **termin** procedure is defined as:

```
procedure termin(file [, string]);
```


file

is a variable of type text.

string

is an optional string value that specifies any special file dependent options that may be used in opening the file. For an explanation of these options, see *RT PC VS Pascal User's Guide*.

Termout — Opens File for Output

The **termout** procedure opens a designated file for output to the terminal.

The **termout** procedure is defined as:

```
procedure termout(file [, string]);
```

file

is a variable of type text.

string

is an optional string value that specifies any special file dependent options that may be used in opening the file. For an explanation of these options, see *RT PC VS Pascal User's Guide*.

Write and Writeln Procedures

The **write** and **writeln** procedures append character strings to a text file. The **writestr** procedure writes data to a string. Details of this procedure can be found in the string manipulation functions in Chapter 8, "Built-In Procedures and Functions." Usually the character strings are generated by converting one or more **write** parameters from their machine representations into external representations.

The **writeln** procedure differs from the **write** procedure only in that **writeln** sends an end-of-line to the output file after the write is complete.

The **write** and **writeln** procedures are defined as:

```
procedure write
procedure write(file, write-parameter {,write-parameter} );

procedure writeln
procedure writeln(file, write-parameter {,write-parameter} );
procedure writeln(f: text);
```

file

is a file variable that refers to the file that is to contain the appended character strings.

Note: If this parameter is omitted, the default file "output" is used.

write-parameter

is one or more parameters used to control the format of the individual elements. The individual elements are described in the following sections.

Write and Writeln Parameters

The **write** and **writeln** procedures can control the format of the individual elements that are written.

Each parameter to **write** or **writeln** is of the form:

```
Form 1  element
Form 2  element:len1
Form 3  element:len1:len2
```

element

is an expression that may be one of the following:

- integer or subrange
- real or shortreal
- char element
- boolean
- string
- packed array of char.

Each of these expressions is described in detail in subsequent sections.

len1

supplies the length of the field into which the data is written. The data is placed into the field justified to the right edge of the field. If *len1* specifies a negative value, the data is justified to the left within a field whose length is `abs(len1)`.

len2

is an expression (Form 3) which may be specified only if the expression is of the type real.

Note: If *len1* is unspecified (Form 1), the default value is used according to the Figure 6-1.

Type of Expression	Default Value of len1
integer	12
real	20 (E notation)
shortreal	20
char	1
boolean	10
string	length (expression)
packed array of char	length of array

Figure 6-1. Element Default Values

Integer Element: The value of the *integer-expression* is converted into a string representation of that expression in base 10. When a field width is specified that is greater than needed, the resulting string is right-justified and placed into the output field. If the *field-width* is too small to contain the resulting character string, the output field is expanded until it can contain the output string. If a negative *field-width* is specified, the resulting string is left justified and placed into the output field.

Scalar Subrange Element: A *write* parameter that is a scalar subrange is handled exactly as the scalar range of which it is a subrange.

Writing Real (Shortreal) Data: Real expressions may be printed with any one of the three operand formats.

1. If *len1* is not specified (Form 1), the result is in scientific notation in a 20 character field.
2. If *len1* is specified and *len2* is not (Form 2), the result is in scientific notation, but the number of characters in the field is the value of *len1*.
3. If both *len1* and *len2* are specified (Form 3), the data is written in fixed point notation in a field with length *len1*. In this form, *len2* specifies the number of digits to the right of the decimal point.

The real expression is always rounded to the last digit to be printed. If *len1* is not large enough to fully represent the number, it is extended appropriately.

Examples:

Call:	Result:
<code>write(3.14159:10);</code>	<code>' 3.142E+00'</code>
<code>write(3.14159);</code>	<code>' 3.14159000000000E+00'</code>
<code>write(3.14159:4);</code>	<code>' 3.1416'</code>

Writing Char Data: The value of *len1* is used to indicate the width of the field in which the character is to be placed. If the *field-width* is not specified, it is assumed to be 1. If the *field-width* is greater than 1, the character is padded on the left with blanks. If the *field-width* is negative, the character is padded on the right.

Writing Boolean Data: The expression *len1* is used to indicate the width of the field in which the boolean is to be placed. If the width is less than 6, either a "t" or "f" is printed. Otherwise, "true" or "false" is sent to the file. The data is placed in the field and justified according to the previously stated rules.

Examples:

Call:	Result:
<code>write(true: 10);</code>	<code>' true'</code>
<code>write(true:-10);</code>	<code>'true '</code>
<code>write(false:2);</code>	<code>'f'</code>

Writing String Data or Packed Array of Char: The expression *len1* is used to indicate the width of the field in which the string is to be placed. The data is placed in the field and justified according to the rules previously stated. If `abs(len1)` is too small to hold the data, the string truncates on the right.

Examples

Example	Result
<code>write ('abcd' : 6)</code>	<code>' abcd,'</code>
<code>write ('abcd' : -6)</code>	<code>'abcd '</code>
<code>write ('abcd" : 2)</code>	<code>'ab'</code>
<code>write ('abcd')</code>	<code>'abcd'</code>

Writing to a File of Any Type

The **write** procedure can also write to a file of any type. A **write** procedure call of the form:

```
write(file, expr1, expr2..., exprn);
```

is equivalent to the sequence:

```
file@ := expr1;
put(file);
file@ := expr2;
put(file);
...
```

```
file@ := exprn;
put(file);
```

where `exprn` is a list of expressions to be written to the file.

Note: The type of each expression in the list must be the same as the type of the elements in the file. Integer subranges are converted to the proper length as needed.

Chapter 7. Program Structure

An RT PC VS Pascal program is a collection of declarations and statements that can be translated by a compilation process into a relocatable object-module. Object modules obtained from other separate compilations can be combined by a linking process into a form suitable for execution.

The collection of declarations and statements may also include compiler directives that control the compilation and do not change the meaning of the program. For a complete list of the RT PC VS Pascal compiler directives, see the *RT PC VS Pascal User's Guide*.

The results of compilations, object modules, are sometimes referred to as ".o" files since this is the normal file name extension for such files. RT PC VS Pascal is very flexible in the mechanisms for creating .o files that are not complete executable programs and combining them in the linking process.

The "external" and "segment" mechanisms can be used for independent compilation of RT PC VS Pascal programs or for linking RT PC VS Pascal to programs written in other languages.

Compilation Units

This section describes the various compilation units and their components in detail. Before this description, however, there is a section containing examples of compilation units with accompanying explanations.

Compilation Unit Programs

Program 1

The following program is complete and can be compiled and executed. It does not make use of any separate compilation.

```
program complete;
  var i: integer;
  begin
    i := 17;
    writeln(i);
  end.
```

Program 2

This example illustrates the call on an external procedure named `getvalue` that is supplied in the linking process in order to make a complete executable program.

```
program missingsomething;
  var i: integer;

  procedure getvalue(var fi: integer); external;

  procedure callme;
  begin
    writeln('I got called!');
  end;

  begin
    getvalue(i);
    writeln(i);
  end.
```

It is possible that this external procedure has been written in FORTRAN, C, or in RT PC VS Pascal. For more information on mixing the languages, see the *RT PC VS Pascal User's Guide*.

Regardless of the origin of `getvalue`, the RT PC VS Pascal system makes no attempt to match parameter types, etc., between the call and the code called. RT PC VS Pascal is satisfied that the external declaration describes the interface and you should ensure that the receiving subroutine is suitable. This method of independent compilation is referred to as "insecure".

The example also contains a procedure "callme" that may be referenced in some other compilation unit as an external. This other compilation unit must not, however, contain a main program since it is not allowed to link together object files containing more than one main program.

Definitions

A "compilation unit" is either a main program or a segment. A complete executable program consists of a single program and zero or more segments.

A "program" is a main program, consisting of all the statements between a program statement and an end statement. The main program is described in more detail in the "Program Heading" section below.

A "segment" is a collection of declarations and statements packaged to make parts of the declarations in the segment public to other parts of the same compilation segment or separate compilation segments. Segments are useful for sharing common code among different programs or as a means to avoid compiling a huge program every time one line is changed. Segments are compiled separately.

A program or segment that uses another segment is known as a "host". A host uses other segments' declarations by naming those units in uses declarations.

Program Heading

The program statement identifies the main program for an RT PC VS Pascal compilation. In RT PC VS Pascal, the *program-heading* is scanned and then ignored. A program has the same form as a procedure declaration except for the heading.

A program is defined as:

program-heading; block.

A *program-heading* is defined as:

program *identifier* [(*program-parameters*)];

identifier

is the program name.

program-parameters

contains one or more identifiers, separated by commas. These parameters are optional.

No global identifiers in the program may have the same name as any of the *program-parameters*.

Segment Module

A "segment module" consists of routines that are linked into the final program prior to execution. It may be compiled as a unit independent of the program module. Data is passed to routines through parameters and external variables. Segments are useful in breaking up large RT PC VS Pascal programs into smaller units.

The global automatic variables of the program module may be accessed in a segment module.

The identifier following the reserved word **program** must be a unique external name. The identifier following the word **segment** may be the same as one of the external routines in the segment or may be a unique external name. Thus, a function called "cosine" could be in a segment called "cosine". An external name is an identifier for a program, segment, **def** or **ref** variable, external routine, **main** procedure or a **reentrant** procedure. The **main** and **reentrant** procedures are described in the "Procedure and Function Declaration" section.

The optional identifier list following the program identifier is not used by RT PC VS Pascal. The identifiers are ignored.

An executable program is formed by linking a program module with segment modules (if any) and with the RT PC VS Pascal execution library and any other specified libraries.

Example

```
segment cosine;
  function cosine
    (x : real) : real; external;
  function cosine;
  var s: real;
  begin
    s := cosine(x);
    cosine := sqrt (1.0 - s*s)
  end;
```

Scope of Identifiers

Declarations introduce program objects, together with their identifiers, that denote these objects elsewhere in a program. The scope of the identifier is the program region in which all uses of an identifier are associated with the same object. Within a compilation unit, such a region is either a segment body or a block body. In the case of a **segment**, the scope is a declaration list. In the case of a **block**, the scope is a statement list preceded by an optional declaration list.

Segment Scope Rule

The scope of an identifier is determined by the context in which it is declared.

A program or a segment is a static construct intended to control the scope of identifiers according to the following rule:

- The scope of an identifier declared at the outermost level of a program is the body of that program.

Block Scope Rules

Procedure or function blocks also control the scope of identifiers. There are similarities and differences between blocks and programs or segments. Like programs or segments, blocks control the scope of identifiers. Unlike programs or segments, blocks control the processing of declarations and determine when the declarations take effect.

The block-structured scope rules are:

- The scope of an identifier declared in the declaration list of a block is the body of that block.
- If the scope of an identifier includes another block, its scope is extended "inward" to include the body of that inner block unless the body contains a re-declaration of that identifier.

- An identifier that is declared as a formal parameter of a procedure or function has as its scope the body of that procedure or function.
- Field selectors are identifiers introduced as part of the definition of a record type for the purpose of selecting fields of records. The scope of a field selector is the record in which it is declared. As with the nesting of procedures, the existence of an inner scope identifier masks the accessibility of any outer identifiers of the same name. Field selectors must be unique within the declaration of a record.
- Identifiers must be unique within the bounds of a given scope.

Standard Files

RT PC VS Pascal supplies three standard files. These files are:

input	is the standard file from which console input can be done by read and readln statements. By default, it is associated with the standard input.
output	is the standard file to which console output is directed by write and writeln statements. By default, it is associated with the standard output.
stderr	is the standard error output file. By default, stderr is directed to the same place as output.

Declarations

Declarations introduce programs objects and their identifiers used in the program. The following declarations types are described in the next few sections.

- label declarations
- def/ref declarations
- static declarations
- value declarations
- space declarations
- constant definitions
- type definitions
- variable declarations

Label Declarations

The **label** declaration part declares all labels (which tag statements) in the statement part of the block. The label-declaration part is defined as:

label *label* {, *label*};

label

is an integer in the range of 0..9999 or an identifier.

Def/Ref Declarations

The **def/ref** declarations are used to declare external variables. External variables are allocated prior to execution and can be accessed from more than one module. All identifiers that are used as external names must be unique in the first eight characters.

If an external variable with a particular name is declared in several modules, a single common storage location is associated with each such variable. An external variable must be declared with identical types in each module.

The **def** declaration specifies that the program loader is responsible for generating the common storage for the variable. The **ref** declaration specifies that storage for the variable is defined in another module. All **ref** declared variables remain unresolved until the encompassing module is compiled and linked with a module in which the variable is declared as a **def** variable.

A **def** or **ref** variable may be declared as local to a routine; the same scope rules apply as for any other declared identifier. However, if the name of the variable is declared in another scope (even in another module) as a **def** or **ref** variable, both occurrences of the variable reference the same storage.

A **def** variable may be initialized at compile-time by the use of a value declaration. Programs which modify **def**, **ref**, or **static** variables are not reentrant.

In procedures a, b, and c in the following examples, the variable "x" references the same storage. However, both the "x" variables declared in segment p and procedure d refer to storage that is separate from the external variable "x".

Examples:

```
segment m;
procedure a; external;
procedure a;
  def x: real;
  begin
    ...
  end;
```

```
procedure b; external;
procedure b;
  def x: real;      { same as x in a }
  begin
    ...
  end;
```



```

segment p;
static x: real;    { local to p    }
procedure c;

    ref x: real;    { same as x in a,b }
begin
    ...
end;

procedure d;
var x: real;       { local to d    }
begin
    ...
end;

```

Static Declarations

The **static** declaration is used to declare **static** variables. The variables declared in this way are allocated prior to program execution and exist for the life of the program's execution.

A **static** variable can be referenced according to the scope rules as described in "Block Scope Rules." Two **static** variables in different scopes are different variables even though they have the same name.

Data in **static** variables that are local to a routine are preserved over separate invocations of the routine. Such a routine called recursively accesses the same instance of each **static** variable.

A **static** variable may be initialized at compile time by the use of a **value** declaration. Programs which modify **static** variables are not reentrant.

Examples

```

static
  sysprint : text;
  x,y:      real;

```

Value Declarations

The **value** declaration is used to specify an initial value for **static** and **def** variables. It is composed of a list of value-assignment statements, separated by semicolons. Pointers may only be initialized to nil by using the **value** declaration.

The assignment statements in a **value** declaration have the same form as the assignment statements in the body of a routine except that all subscripts and expressions must be able to be evaluated at compile time.

Example:

```
{Value Declaration      }
type
  complex = record
    re,im: real;
  end;
  vector  = array[1..7] of integer;

static
  c: complex;
  v: vector;
  v1: vector;

def
  i : integer;
  q : array[1..10] of complex;

value
  c      := complex(3.0,4.0);
  v      := vector(1,0:5,7);
  v1     := vector(,,4);
  v[2]   := 2;
  v[3]   := 3*4-1;
  i      := 0;
  q[1].re := 3.1415926 / 2;
  q[1].im := 1.414;

{ The above assignments take place at compile time }
```

```

{ Initialization of a three-dimensional array }
type
    cube = array[1..10,1..10,1..10] of real;

static
    block    : cube;

{ The following assignments take place at compile time }
value
    block    := cube( ( (0.0:10):10 ):10 );

```

If a **def** variable is initialized with a **value** declaration in one module, a **value** declaration can not be used on that variable in another module. The compiler does not check this violation; however, a diagnostic message is generated when the modules are combined into a single load module by the system loader. Diagnostic messages are explained in *RT PC VS Pascal User's Guide*.

Space Declarations

The **space** declaration is used to represent a storage area of a fixed number of bytes with a data component type. The space declaration allows the offset of the component type to vary within the storage area.

The **space** declaration is defined as:

space [{*constant-expr*}] of {*type*}

constant-expr

represents the size of the storage area (in bytes) associated with the type.

type

is any type except a file type.

A variable declared as a **space** occupies the number of bytes indicated in the length specifier of the type definition.


```
var s: space[1000] of integer;
```

In this example, the variable "s" occupies 1000 bytes, and the component data type is integer.

A variable declared with **space** is treated as a variable of the component type. A **space** variable is accessed by following the variable's name with an integer index expression enclosed in square brackets. The index represents the appropriate offset (as described in the following paragraph) within the space variable's storage where the accessed data resides. The offset is specified with an origin of zero.

RT PC VS Pascal allows the displacement index of the component variable of type **space** to take on any integer value from zero to the maximum declared size of the variable. The architecture of the RT PC requires that the component variable of type **space** be aligned on 1, 2, or 4 byte boundaries depending upon the number of bytes required by the component type. This means that a 1 byte component type (for example, of type char) is aligned on 1 byte boundaries, 2 byte components (for example, packed array[1..2] of char) are aligned on 2 byte boundaries, and any type requiring 3 or more bytes of storage are aligned on 4 byte boundaries. As a result of these constraints, the integer index value is mapped to the appropriate boundary required by the component type: the index value does not necessarily represent the true byte offset within the space variable.

Examples

```
{In this example, references to svar[0], svar[2],  
svar[3] are mapped to the zero offset, svar[0],  
within the variable svar. }
```

```
var  
  svar : space[40] of integer;
```

Note: As long as a space type variable is used to access whole components on aligned boundaries, the actual mapping of these components into memory is transparent to the program. Attempts to use a space type variable for byte shifting using index values to offset on unaligned boundaries produces results that are undefined.

```
{Using this declaration: }
```

```
type
  rec = record
    a : 0..127;
    b : char;
    c : integer;
    d : real;
  end;
var
  srec = space[40] of rec;
  i : integer;
```

```
{ and the following code }
```

```
with srec[0] do begin
  a := 127;
  b := 'c';
  c := 32767;
  d := 333.33;
end;
i := srec[0].a;           (i is set to 127)
i := srec[4].a;          (i is set to the contents of the
                          4th byte of srec on the RT PC )
```

The actual memory representation of the component type of a space variable can give different results depending upon the implementation and architecture involved.

An element of a **space** may not be passed as a **var** parameter to a routine. However, an element may be passed as a **const** or **value** parameter.

Space Referencing

A component of **space** is selected by placing an index expression, enclosed within square brackets, after the **space** variable (just as in array references).

The indexing expression must be of type integer (or a subrange). The value of the index is the offset within **space** at which the component is accessed. The unit of the index is the byte. The index is always based upon a zero origin; the index range of the space is from zero to one less than the value of the constant expression. The component is the space base type.

Example:

```
var
    {declare a space variable with index range 0..99 }
record
    a,b : integer;
end;

begin
    {base record begins at offset 10 within space }
    s[10].a := 26;
    s[10].b := 0;
end;
```

Constant Definitions

The **constant** definition part declares all constant names and their associated values that are local to the procedure or function definition. It allows for the assignment of the identifiers that are used as synonyms for constant expressions. The type of a constant identifier is determined by the type of the expression in the declaration.

The constant definition part is defined as:

const *constant-definition-list*;

constant-definition-list

contains one or more *constant definitions* separated by semicolons.

Type Definitions

The **type** definition part contains all the type definitions that are local to the procedure or function definition. A **type** is defined as:

```
type type-definition-list;
```

type-definition-list

contains one or more type definitions separated by semicolons.

Variable Declarations

The **variable** declaration part contains a definition of all the variables that are local to the procedure or function. The variable declaration part is defined as:

```
var variable-declaration-list;
```

variable declaration list

contains one or more variable declarations separated by semicolons.

Procedure and Function Declarations

A **procedure declaration** or a **function declaration** associates an identifier, which is the procedure or function name, with a collection of declarations and statements. An RT PC VS Pascal statement can then cause the execution of that procedure by giving its name in a procedure reference statement.

A **function declaration** is similar to a **procedure declaration** except that it has the additional capability of being able to compute and return a value, known as the value of the function. A function is referenced by giving its name in an expression when the value of the function appears as a factor in that expression.

The type of value that a function returns is specified when the function is declared. The return value of the function is the value last assigned to its function identifier before a return is made from the function. Returning from a function without assigning a value to the function designator (for the current activation of the function) produces an undefined result.

Using a procedure or function identifier within the body of that procedure or function implies recursive execution of that procedure or function. However, recursive execution does not occur when the function identifier appears on the left side of an assignment statement. This indicates assignment to the function variable rather than recursive activation.

Procedure Definition

The procedure declaration is defined as:

procedure-heading
block

procedure-heading

specifies the identifier that names the procedure and any formal parameters for that procedure. The procedure-heading is defined as:

procedure identifier [(*formal-parameters*)] ; [*attribute* ;]

identifier

is the name given to the procedure.

formal-parameters

are value parameters, variable parameters, or procedure or function parameters. These parameters are described in the "Parameters for Procedures and Functions" section in this chapter.

attribute

indicates a separately compiled routine or a routine assembled by a compiler other than RT PC VS Pascal. The attributes that may be specified are:

main
reentrant
FORTRAN
external
forward

These attributes are optional. The "Calling Attributes" section in this chapter contains a more detailed explanation of these attributes.

block

may consist of the following parts:

label-definition-part
constant-definition part
type-definition-part
variable-declaration-part
procedure and function-declaration part
statement-part

All the definition and declaration parts above are optional, with the exception of the *statement-part*. The *statement-part* is defined as:

begin *statement-list* end.

statement list

contains one or more statements separated by semicolons.

The usual declare-before-usage rules apply; that is, the declarations are processed in the order in which they occur, as contrasted with processing all of the constant sections followed by all of the type sections, and so on.

"Forward" pointer-type references are resolved in each type definition part without regard to definitions that occur in later declaration sections. This feature facilitates the inclusion of files of declarations that are logically complete without segregating declared items based on whether they are constants, types, or variables.

Function Definition

The function declaration is defined as:

function-heading
block

The function-heading is defined as:

function *identifier*: *result-type*;{*attribute*;}
or
function *identifier*: *result-type* (*formal-parameters*);{*attribute*;}
or
function *identifier*: *result-type* (*formal-parameters*);{*attribute*};

identifier

is the name given to the procedure.

result-type

is any type except file.

attribute

indicates a separately compiled routine or a routine assembled by a compiler other than RT PC VS Pascal. The attributes that may be specified are:

main
reentrant
FORTRAN
external
forward

These attributes are optional. The "Calling Attributes" section in this chapter contains a more detailed explanation of these attributes.

formal-parameters

are arguments used to process different sets of data. These arguments are described further in the "Parameters for Procedures and Functions" section in this chapter.

block

may consist of the following parts:

label-definition-part
constant-definition part
type-definition-part
variable-declaration-part
procedure and function-declaration part
statement-part

Note: Only the *statement-part* is required.

Parameters for Procedures and Functions

Parameters or arguments provide a dynamic substitution method so that a procedure or function can process different sets of data in different activations.

The formal parameters are defined as:

formal-parameter {;formal-parameter}

formal-parameter
is one of the following:

parameter-group
var *parameter-group*
procedure-heading
function-heading

The *parameter-group* is defined as:

identifier {,identifier}:type-identifier

There must be agreement in both number and type between the formal parameters declared in a procedure or function heading and the actual parameters supplied when the procedure or function is activated.

The procedure or function heading declares a list of formal parameters. These are "dummy" variables that are assigned values when the procedure or function is activated.

A reference to the procedure or function supplies a list of actual parameters that are substituted for the formal parameters that then become local variables initialized to the value of the actual parameters.

The formal parameters are:

- value parameters
- variable or reference parameters
- procedure parameters
- function parameters

- pass by const parameters
- conformant string parameters

A parameter group without a preceding specifier implies that the parameter is a value parameter.

Value Parameters

Value parameters are those whose formal declaration has no symbol marking them as one of the other three forms. The corresponding actual parameter must be an expression.

In the body of the procedure or function, the formal parameter is initialized to the value of the expression at the time the procedure or function is activated. The formal parameter is then just like a local variable. The value of the formal parameter may be changed by assignment (the actual parameter remains unchanged).

Variable Parameters

Variable parameters (also known as reference parameters) are those whose declarations start with the term **var**. The actual parameter must be a variable of a type that is identical to that of the formal parameter. The formal parameter directly represents and can change the actual parameter's value during the entire execution of the procedure or function.

The **var** actual parameters must be distinct actual variables. It is a programming error to supply the same variable to more than one actual parameter in a procedure or function reference.

All index computations, field selection, and pointer referencing are done at the time the procedure or function reference is made.

Note: Fields of a packed record or elements of a packed array may be passed as **var** parameters in IBM Mode only.

Pass by Const Parameters

Parameters passed by **const** may not be altered by the called routine. In addition, the actual parameter value should not be modified if the call to the routine is not completed. If an attempt is made to alter the actual parameter while it is being passed by **const**, the result is not defined. This method could be called *pass by read only reference*. The parameters appear as constants from the called routine's point of view.

An expression, variable, or constant may be passed by **const**. Fields of a packed record and elements of a packed array may also be passed. The use of the **const** reserved word in a parameter indicates that the parameter is to be passed by this mechanism. With parameters which are structures (such as strings), passing by **const** is usually more efficient than passing by value.

Conformant String Parameters

The *conformant string parameter* can be used to call a procedure or function and pass a string whose declared length does not match that of the formal parameter.

The *conformant string parameter* is a pass by **const** or pass by **var** parameter with a type specified as string without a length qualifier. Strings of any declared length then conform to such a parameter. The **maxlength** function can be used to obtain the declared length.

Examples:

```
procedure translate
  (var   s      : string;
   const table: string);

var
  i   : 0..32767;
  j   : 1..ord(highest(char))+1;
```

```

begin
  for i := 1 to length(s) do
    begin
      j := ord(s [ i ]) + 1;
      if j > length(table) then
        s [ i ] := ' '
      else
        s [ i ] := table [ j ];
      end;
    end;
  end;
end;

```

Procedure and Function Parameter Examples

Procedure and function parameters are the names and parameter lists of procedures or functions that can be referenced by the current procedure.

These parameters are indicated by the term **procedure** or **function** in the formal parameter declarations. Such procedures and functions are known as "parametric". Actual parameters to parametric procedures and functions must be of identical type to those declared in the formal parameter declarations.

Examples:

```
{ A procedure with only value parameters }
```

```

procedure bythebook(chapter, verse: integer);
begin
  chapter := 1;           { Does not change the caller's
                           version of chapter }
end;

```

```
{ A procedure with variable parameters }
```

```

procedure change(var winds: integer);
begin
  winds := 76;           { Changes the caller's version }
end;

```



```

{ The baker function }

function baker(m, n: integer):integer;
begin
    if m = 0 then
        baker := n + 1
    else if n = 0 then
        baker := baker(m - 1, 1)
    else
        baker := baker(m - 1, baker(m, n - 1))
    end;

{ Parametric function parameter }

function integrate(lo, hi: real;
function what(x: real):real): real;

var
    start: integer;
    finish: integer;
    point: integer;
    current: real;
    sum: real;

begin
    start := trunc(lo);
    finish := round(hi);
    sum := 0.0;

    for point := start to finish do
    begin
        current := point;
        sum := sum + what(current);
    end;
    Integrate := sum / (finish - start);
end;

```

Calling Attributes

These calling attributes may be used with RT PC VS Pascal procedures:

- **main** attribute
- **reentrant** attribute
- **FORTTRAN** attribute
- **external** and forward attribute

Main Attribute

The **main** attribute is used to identify an RT PC VS Pascal procedure that may be invoked as if it is a main program. It is sometimes desirable to invoke an RT PC VS Pascal procedure from another language routine, for example FORTRAN or C language. In this case, it is necessary that certain initializing operations be performed before actually executing the RT PC VS Pascal procedure. The **main** directive specifies that these actions are to be performed.

The following notes apply to the use of the **main** attribute.

- only procedures may have the **main** attribute
- a procedure that is declared as **main** must have its body located in the same module
- the execution of a **main** procedure can not be reentrant
- the **main** attribute may only be applied to procedures in the outermost nesting level

Reentrant Attribute

The **reentrant** attribute is used to identify an RT PC VS Pascal procedure that may be invoked as if it were a **main** procedure. In addition, invocations of these procedures are reentrant.

In order to achieve this additional feature, several rules should be followed. The first parameter of a procedure defined with the **reentrant** attribute must be an integer passed by **var**. Prior to the very first call from another language program, this variable must be initialized to zero (0). On subsequent calls the same variable must be passed back unaltered. RT PC VS Pascal sets the variable on the first call and needs that value on the subsequent invocations. The same procedure does not need to be called each time; different procedures can be called. However, the variable must continue to be passed on each call.

Note: All RT PC VS Pascal internal procedures and functions are reentrant. The **reentrant** attribute is used to specify a procedure that is both reentrant and invocable from outside the RT PC VS Pascal execution environment.

FORTTRAN Attribute

The **FORTTRAN** attribute instructs RT PC VS Pascal to utilize exactly the same calling conventions employed by FORTRAN. This limits the form of the parameter list: a parameter cannot be passed by **var** or by **const**. If a parameter is passed by **var** or **const**, the FORTRAN subprogram must not modify the parameter.

External and Forward Attributes

An RT PC VS Pascal module can use routines that are separately compiled or assembled in languages other than RT PC VS Pascal. To use an external routine, the host must make a procedure or function declaration for that external routine just as if it is an RT PC VS Pascal routine that is declared in this compilation unit or another compilation unit. The declaration is then followed by the external attribute to indicate that the body routine does not appear in the current compilation unit. This attribute may also be used for Pascal routines that are separately compiled. External routines must

conform with the RT PC VS Pascal calling conventions and data representation methods as defined in the *RT PC VS Pascal User's Guide*.

RT PC VS Pascal normally dictates that procedures and functions be declared before they can be referenced. There are cases when program layout makes this impossible, and a procedure or function must be referenced before it can be declared. The forward attribute indicates that the particular procedure or function declaration consists only of the header and that the body of that procedure or function appears later in the program source text, possibly after it is referenced. Therefore, a forward-declared procedure or function is actually declared in two distinct parts: its header or prototype is declared with the forward attribute before any reference is made to it; at some later point in the program source text, its body is declared. At this later point, the formal parameter section must not appear.

Chapter 8. Built-In Procedures and Functions

RT PC VS Pascal supplies a number of standard or "built-in" procedures and functions. The standard procedures and functions fall into logically related groups:

- string manipulation routines which handle the RT PC VS Pascal dynamic string types
- memory management routines which deal with dynamic memory allocation and de-allocation
- arithmetic routines
- boolean routines
- value conversion routines
- miscellaneous low-level routines
- control routines
- additional standard routines.

String Manipulation Routines

This section describes the facilities that manipulate string data types in RT PC VS Pascal. For purposes of this section, string data types are those declared `string(n)` for some *n*, not packed array[1..*n*] of char. The type "stringtype" used hereafter should be read as matching any type declared `string(n)`. The following list summarizes all the string manipulation routines:

compress	replaces multiple blanks with one blank.
delete	deletes characters from a string.
index	scans for a pattern within a string and returns the starting index of the first instance of the pattern in the source string.
length	determines the current dynamic length of a string.
lpad	pads or truncates the string variable on the left.
ltrim	removes the leading from a string.
maxlength	returns the declared static length of a string.
readstr	converts a string to a different type.
rpad	pads or truncates the string variable on the right.
substr	returns specified substring from a string.
token	scans a string and returns it as an alpha.
trim	removes the trailing blanks from a string.
writestr	converts an expression into character data.

Compress — Replaces Multiple Blanks

The **compress** function replaces multiple blanks with a single blank. The **compress** function is defined as:

```
function compress(  
  const source : string)  
  : string;
```

source

is a string expression to be compressed.

Example:

```
compress('a b cd ') yields 'a b cd '
```

Delete — Deletes Characters from String

The **delete** function removes a specified number of characters from a string. This function removes *size* characters from *destination* starting at the position specified by *index*.

The **delete** function is defined as:

```
function delete(destination: stringtype;  
               index: integer;  
               {size: integer}):string;
```

destination
is a string.

index
is the starting position of scan.

size
is the number of characters to be deleted. This parameter is optional.

If *index* is greater than **length** (*destination*), there is no action taken. If either *index* or *size* is negative or zero, there is no action taken. If *index* + *size* is greater than **length** (*destination*), **delete** removes all characters from *index* up to the end of the *destination* string. If the *size* parameter is omitted, the string is truncated beginning at the position specified by *index*.

Example:

```
var
  large: string(100);
begin
  large := 'A long exhausting cross-country hike';
  large := delete(large, 8, 11);
  writeln(large);
end;
```

This generates the output:

A long cross-country hike

Index — Returns Starting Index

The **index** function compares the second parameter against the first and returns the starting index of the first instance where *lookup* begins in *source*. If there are no occurrences, a zero is returned.

The **index** function is defined as:

```
function index(
  const source : string;
  const lookup : string)
  : 0..32767;
```

source

is a string that contains the data to be compared with the *lookup* string.

lookup

is the data to be looked up in the *source*.

Examples:

```
var
  s : string;
...
s:= 'abcabc';
...
index(s,'bc') yields 2
index(s,'x') yields 0
```

Length — Determines String Length

The **length** function is an integer function that returns the length of a string expression. It returns an integer value which is the dynamic length of the string *source*. The length of the string '' is zero.

The **length** function is defined as:

```
function length (source: stringtype): integer;
```

Examples:

```
alphabet := 'abcdefghijklmnopqrstuvwxyz';
writeln(length(alphabet), ' ',
        alphabet(1), ' ',
        alphabet(length(alphabet)), ' ',
        length(''));

```

The following output is displayed:

```
26  a  z  0
```

Lpad Procedure — Pads or Truncates String on the Left

The **lpad** procedure pads or truncates the string variable on the left. If the length of the string is longer than the final length, the effect is to truncate characters on the left. If the length of the string is less than the final length, the effect is to extend the string to the left with the character specified.

The **lpad** function is defined as:

```
procedure lpad(  
  var s      : string;  
    i      : integer;  
    c      : char);  
external;
```

s
is the string to be padded or truncated.

i
is the final length of *s*.

c
is the pad character.

Ltrim — Removes Leading Blanks

The **ltrim** function returns the parameter value with all leading blanks removed. The **ltrim** function is defined as:

```
function ltrim(  
  const source : string)  
    : string;
```

source

is the string to be trimmed.

Example:

```
ltrim(' a b ') yields 'a b '  
ltrim('      ') yields ''
```

Maxlength — Returns Maximum Length of a String

The **maxlength** function returns the maximum length of the parameter string. The value is in the range 0..32767.

The **maxlength** function is defined as:

```
function maxlength(  
  s      : string)  
  : 0..32767;
```

s
is a string valued expression.

Example

```
program maxlen;  
  var title : string (255);  
      test1 : string (62);  
      x     : integer;  
begin  
  title := 'left-side middle part right-side';  
  test1 := 'maxlength should return a 62';  
  x     := maxlength (title);  
  writeln (x);  
  x     := maxlength (test1);  
  writeln (x);  
end;
```


This generates the output:

255 62

Readstr — Converts a String to Another Type

The **readstr** procedure reads character data from a source string into one or more variables. The actions of **readstr** are identical to that of **read** except that the source data is extracted from a string expression instead of a text file. The **readstr** function is especially useful for converting a string to a different type.

The **readstr** function is defined as:

```
procedure readstr(  
  const s : string;  
  v : variable);
```

s

is a string expression that is to be used for input.

v

is a list of one or more variables, which must be one of the following types:

- integer (or subrange)
- char (or subrange)
- real
- shortreal
- string
- packed array of char

As in the **read** procedure, variables may be qualified with a field length expression.

Note: The results are unpredictable when reading a string greater than the static length.

Examples:

```
var s : string(10);
    s1 : string(10);
    i : integer;
    .
    .
    i :=20
    s :='ABCDEFGHIJ';
    readstr (s,s1:i);
```

The above example will produce unpredictable results.

```
var
    i,j: integer;
    s : string(100);
    s1 : string(100);
    ch : char;
    cc : packed array[1...10] of char;
    .
    .
    s := '36 245abcdefgghijk';
    readstr(s,i,j:3,cj,cc:5,s1);
```

The variables would be assigned:

i	36
j	24
ch	'5'
cc	'abcde '
s1	'fghijk';
length(s1)	6

Rpad Procedure — Pads or Truncates String on the Right

The **rpad** procedure pads or truncates a string variable on the right. If the length is greater than the final length, the effect is to truncate characters on the right. If the length is less than the final length, the effect is to extend the string with the character specified to the right of the string.

The **rpad** procedure is defined as:

```
procedure rpad(  
  var s      : string;  
      l      : integer;  
      c      : char);  
external;
```

s
is the string to be padded or truncated.

l
is the final length of *s*.

c
is the pad character.

Substr — Returns Substring

The **substr** function returns a substring from a specified string *source*. The *start* parameter specifies the starting position in the source string from which the substring is to be extracted. The first character of the source string is at position 1. The *len* parameter determines the length of the substring. If the length is omitted, the substring returned is the remaining portion of the source string from the position indicated by *start*.

The **substr** function is defined as:

```
function substr(  
  const source : string;  
  start : integer;  
  {len : integer}): string;
```

source

is the string expression from which a substring is returned.

start

is an integer expression that designates the first position in the *source* to be returned.

len

is an integer expression that defines the number of characters to be returned. This is an optional parameter.

The value of `start+len-1` must be less than or equal to the current length of the string.

Examples:

```
substr('abcde',2,3) yields 'bcd'  
substr('abcde',1,3) yields 'abc'  
substr('abcde',4) yields 'de'  
substr('abcde',1) yields 'abcde'  
substr('abcde',2,5) is an error
```

Note: The **substr** and **copy** facilities are functionally identical.

Token — Returns String as Alpha

The **token** procedure scans the *source* string looking for a token and returns it as an **alpha**. The starting position of the scan is passed as the first parameter. This parameter is changed to reflect the new position for scans on subsequent calls. Leading blanks, multiple blanks, and trailing blanks are ignored. If there is no token in the string, *pos* is set to `length(source)+1`, and *result* is set to all blanks.

A **token** may be one of the following:

- An RT PC VS Pascal identifier is 1 to 16 alphanumeric characters, a "\$", or an underscore. The first letter must be alphabetic or a "\$".
- RT PC VS Pascal unsigned integer.
- A special symbol:

+	-	*	/	->	@	¢
=	<>	<	<=	>=	>	!
()	[]	'	"	%
	&	&&		~	~=	#
:	;	:=	.	/	..	
{	}	(*	*)	/*	*/	

The **token** procedure is defined as:

```
procedure token(  
  var pos : integer;  
  const source : string;  
  var result : alpha);
```

pos

is the starting index in *source* of the location to look for a token. It is set to the index of the location to resume the search on the next use of **token**.

source

is a string that contains the data from which the token is to be extracted.

result

is the alpha variable which is returned with the token.

Example:

```
i := 2;  
token(i, ' ', testpg+, result)
```

In this example, the "i" is set to the first position after the token is found (position 8). The token that is found is set to "testpg" in an alpha with padding on the left. The **token** returns the same if "i" is set to 3, that is, leading blanks are ignored.

Trim — Removes Trailing Blanks

The **trim** function returns the parameter value with all trailing blanks removed. The **trim** function is defined as:

```
function trim(  
  const source : string  
  : string;
```

source

is the string to be trimmed.

Example:

```
trim(' a b ') yields ' a b'
```


Writestr — Converts Data into Strings

The **writestr** procedure converts expressions into character data and stores the data into a string variable. The actions of **writestr** are identical to **write**, except that the target of the data is to a string rather than to a text file. The **writestr** function is especially useful for converting data into string format.

The **writestr** function is defined as:

```
procedure writestr(  
  var s : string;  
  e : integer);
```

s
is a string variable.

e
is an expression of one of the following types:

- integer (or subrange)
- char (or subrange)
- real
- shortreal
- boolean
- string
- packed array [1..n] of char

RT PC VS Pascal accepts a special parameter format which allows you to specify a length for the result.

As in the case of **write**, the expressions being converted may be qualified with a field-length expression.

Note: The results are unpredictable when writing to a string that has a dynamic length greater than its static length.

Example:

```
var
  i,j: integer;
  s  : string(100);
  r  : real;
  ch : char;
.
.
i  := 10; j := -123;
r  := 3,14159;
ch := '*';
writestr(s,i:3,j:5,'abc',ch,r:5:2);
```

The variable s would be assigned:
' 10 -123abc* 3.14'

Storage Allocation Routines

Dynamically allocated storage is held in a common storage pool known as a "heap". Storage is allocated from that pool by using the **new** procedure.

RT PC VS Pascal handles memory de-allocation with the **mark** and **release** procedures.

For more information on memory usage, see the *RT PC VS Pascal User's Guide*.

dispose	is responsible for freeing or releasing storage back to the common storage pool.
mark	provides a means to "remember" the current top of the heap.
new	is responsible for allocating storage.
release	releases memory from a previously marked point.

Dispose — Disposes of Allocated Storage

The **dispose** procedure frees (or de-allocates) dynamically allocated storage. The **dispose** procedure frees the allocated storage referenced by the pointer variable *p*. Upon return from **dispose**, the pointer *p* contains the value "nil". An attempt to use the **dispose** procedure with a pointer variable that contains a nil value results in "no operation" and is ignored.

If **new** is used to allocate a variable with a specific variant, **dispose** should be called with exactly the same variant or the heap is likely to be corrupted.

The **dispose** procedure is defined as:

```
procedure dispose (p);
```

Mark — Marks Position of Heap

The **mark** procedure is used to "remember" the current position of the top of the heap. The **mark** and **release** procedures are used together to de-allocate memory and return the top of the heap to a previously marked point. For example, a procedure might **mark** the heap upon entry, allocate large numbers of variables, and then, just prior to exiting, **release** all the allocated memory.

Such a situation might occur, for instance, in allocating the local symbol table for an assembly unit. At the end of the unit, all the local labels need to disappear. The **mark** and **release** provide a convenient method for disposing of storage in bulk.

The **mark** procedure is defined as:

```
procedure mark(HeapPointer: @anything);  
  
or  
  
procedure mark(HeapPointer: →anything);
```

The *HeapPointer* must be a pointer. The pointer type is irrelevant, but conventionally it is a pointer to an integer. The *HeapPointer* should not be used for any purpose other than as a **mark** or **release** pointer.

Note: See the description of the **release** facility for an example of both **release** and **mark**.

New — Allocates Storage

The **new** procedure allocates dynamically available storage. If *p* is a variable of type pointer to *t*, **new**(*p*) allocates storage for a variable of type *t* and assigns a pointer to that storage to the variable *p*.

There are three forms of the **new** procedure reference. The first definition allocates a new variable *v* and assigns the pointer reference of *v* to the pointer variable *p*. If the type of *v* is a variant record, storage is allocated for the largest variant of the record.

```
procedure new(p);
```

Example

{This is an example of the first form of the new procedure.}

```
type
  age = 0..100;
  recp = @rec;
  rec =
    record
      name :string (30);
      case how_old: age of
        0..18
          (father: recp);
        19..100;
          (case married: boolean of
            true: (spouse: recp);
            false: ())
          )
    end;

var
  p      : recp;
  ...
begin
  ...
  new (p,18);
  with p@ do begin
    name := 'J. B. Smith, Jr';
    how_old := 18;
    new(father,54,true);
    with father@ do begin
      name := 'J. B. Smith';
      how_old := 54;
      married := true;
      new(spouse,50,true);
      ...
    end {with father@};
  end {withp@};
  ...
end;
```

Storage for a specific variant can be allocated by using the second form of the new procedure.

```
procedure new(p, t1, t2, ..., tn);
```

This definition allocates a variable of the variant, with tag fields $t_1 \dots t_n$. The tag fields must be listed contiguously and in the order of their declaration in the variant record type definition.

If **new** is used to allocate storage for a specific variant record, a subsequent call to **dispose** must use exactly the same variant. Any mismatch between the variants specified on the call to **new** and those on the **dispose** call can damage the integrity of the heap.

If **new** fails to allocate the requested storage (usually because the storage is not available), the pointer variable *p* contains the value "nil" upon return from the procedure. Failure to obtain storage results in a run-time error.

Example:

{This is an example of the second form of the new procedure.}

```
type
  linkp = @link
  link  = record
    name: string (30);
    next: linkp;
  end;
```

```
var
  p,
  head : linkp;
  ...
```



```

begin
  ...
  new(p);
  with p@ do
    begin
      name := '';
      next := head;
    end;
  head := p;
  ...
end;

```

The third form of the **new** procedure allocates a string whose maximum length is known only during program execution. The amount of storage to be available for the string is defined by the required second parameter.

procedure new (*p,i*);

p
is a string pointer.

i
is an integer value expression.

Release — Releases Allocated Memory

The **release** procedure is used to cut the heap back to a point previously marked. As with the **mark** procedure, *HeapPointer* is a pointer of any type, but it is conventionally a pointer to an integer. The **release** procedure cuts the heap back to the place indicated by *HeapPointer*. The *HeapPointer* must be properly initialized by a previous call to **mark**. Both **mark** and **release** must be matched properly.

When a heap is freed, all dynamic variables which were allocated from the heap are also freed. As a result, **release** is a method of disposing of many dynamic variables at one time. The **release** procedure sets its parameter variable *p* to nil.

The **release** procedure is defined as:

```
procedure release(HeapPointer: @anything);  
or  
procedure release(HeapPointer: ->anything);
```

Example

```
{ This example illustrates both mark and release }  
type  
  markp = @integer;  
  linkp = @link;  
  link = record  
    name: string(30);  
    next: linkp;  
  end;  
  
var  
  p      : markp;  
  q1,  
  q2,  
  q3      : linkp;  
begin  
  ...  
  mark(p);  
  ...  
  new(q1);  
  new(q2);  
  new(q3);  
  ...  
  { frees q1, q2, and q3 }  
  release(p);  
  ...  
end;
```

Arithmetic Routines

RT PC VS Pascal provides the following routines for performing mathematical functions.

abs	computes absolute value of its argument.
arctan	computes arctangent of the argument.
cos	computes cosine of the argument.
exp	computes exponential of the argument.
ln	computes natural logarithm of the argument.
random	computes a random number.
sin	computes sin of the argument.
sqr	computes square of the argument.
sqrt	computes square root of the argument.

Abs — Computes Absolute Value

The **abs** function computes the absolute value of its argument x . The absolute value of a number is always positive or zero. The type of the result is the same as the type of x , which must be integer, real, or shortreal.

The **abs** function is defined as:

```
function abs (x);
```


Arctan — Computes Trigonometric Arctangent

The **arctan** function computes the trigonometric arctangent of the argument x . The type of x may be integer, real, or shortreal. The return type of **arctan** is always real or shortreal. It returns a value in radians.

The **arctan** function is defined as:

```
function arctan (x);
```

Cos — Computes Trigonometric Cosine

The **cos** function computes the trigonometric cosine of the argument x . The type of x may be integer, real, or shortreal. The return type of **cos** is always real or shortreal. The argument is in radians.

The **cos** function is defined as:

```
function cos (x);
```

Exp — Computes Exponential of Value

The **exp** function raises the base of the natural logarithm e to the power x , that is, e^x . The type of x may be integer, real, or shortreal. The return type of **exp** is always real or shortreal.

The **exp** function is defined as:

```
function exp (x);
```

Ln — Computes Natural Logarithm of Value

The **ln** function computes the natural logarithm of the argument x . The type of x may be integer, real, or shortreal. The return type of **ln** is always real or shortreal. It is an error to supply an argument less than or equal to zero.

The **ln** function is defined as:

```
function ln (x);
```

Random — Computes a Random Number

The **random** function returns a pseudo random number in the range >0.0 and <1.0 . The parameter s is called the "seed" of the random number and is used to specify the beginning of the sequence. The **random** function returns the same value when called with the same non-zero seed. If a seed value of 0 is passed, **random** returns the next number generated from the previous seed. Therefore, the general way to use this function is to pass a non-zero seed on the first invocation and a zero value thereafter.

The **random** function is defined as:

```
function random (s);
```

s

is an expression that evaluates to an integer value.

Sin — Computes Trigonometric Sine

The **sin** function computes the trigonometric sine of the argument x . The type of x may be integer, real, or shortreal. The return type of **sin** is always real or shortreal. The argument is in radians.

The **sin** function is defined as:

```
function sin (x);
```

Sqr — Computes Square of a Number

The **sqr** function computes the square of x . In effect, it multiplies x by x . The type of the result is the same as the type of x , which must be integer, real, or shortreal.

The **sqr** function is defined as:

```
function sqr (x);
```


Sqrt — Computes Square Root of Value

The **sqrt** function computes the square root of the argument x . The type of x may be integer, real, or shortreal. The return type of **sqrt** is always real or shortreal. It is an error to supply an argument less than or equal to zero.

The **sqrt** function is defined as:

```
function sqrt (x);
```

Boolean Attribute Routine

This section describes the boolean attribute:

odd tests for odd or even.

Odd — Tests an Integer for Odd or Even

The **odd** function returns a boolean value indicating whether the argument is odd or even. The type of the argument x must be an integer. The result is true if x is an odd number or false if x is an even number.

The **odd** function is defined as:

```
function odd (x);
```

x

is a value parameter which must be an integer.

Value Conversion Routines

This section describes the predefined routines which perform conversions from one data type to another. The following list briefly summarizes these conversion functions.

chr	converts an integer to a character.
float	converts an integer to a real.
itohs	converts an integer into a hexadecimal string.
ord	converts a value of an ordinal type to an integer.
pack	copies elements from an unpacked array to a packed array.
round	rounds its argument so that the result is of type integer.
scalar conversion	converts an integer to an enumerated scalar.
str	converts a char to a string.
trunc	truncates its argument to the nearest integer.
unpack	copies elements from a packed array to an unpacked array.

Chr — Converts Integer to Character

The **chr** function converts its argument *x* to a character. The argument *x* must be an integer. The result type of **chr** is the character whose ordinal number is *x*. The argument must therefore be in the range 0..255 for **chr** to return a valid result.

The **chr** function is defined as:

```
function chr (x);
```

x
is an integer value.

Float — Converts Integer to Real

The **float** function converts an integer to a real. RT PC VS Pascal converts an integer to a real implicitly if one operand of the arithmetic or relation operator is real and the other is integer. This function is useful in making the conversion explicit in the program.

The **float** function is defined as:

```
function float (i);  
or  
function float (i:integer expression) :real;
```

i
is an integer valued expression.

Itohs Function — Converts Integer to Hexadecimal String

The **itohs** function converts an integer into a hexadecimal string. This function converts the parameter into a string that contains the hexadecimal representation of the integer.

The **itohs** function is defined as:


```
function itohs(  
  i :integer)  
  :string(8);  
external;
```

i
is the value to be converted.

Ord — Converts Type to Integer Value

The **ord** function returns an integer that is the ordinal number of the argument *x* in the set of values defined by the type of *x*. The argument *x* can be any non-floating point scalar or a pointer. If the argument *x* is a pointer, the **ord** function will return the machine address of *x*@.

The **ord** function is defined as:

```
function ord (x);
```

x
is a value parameter of any non-floating point scalar.

Example:

```
var  
  one_letter : char;  
  converted : integer;  
  
begin  
  one_letter := 'm';  
  converted := ord(one_letter);
```

At the end of this program fragment, the variable *converted* has the value 109 since that is the ordinal position of lowercase M in the ASCII character set.

Pack — Copies Unpacked Array to Packed Array

The **pack** procedure copies elements from the unpacked source array, starting with a specified element, to the packed target array. The type of the elements of the two arrays must be identical.

The **pack** procedure is defined as:

```
procedure pack(  
  const source : array__type;  
    index : index__of__source;  
  var target : pack__array__type);
```

source
is an array.

index
is an expression which is compatible with the index of *source*.

target
is a packed array variable.

Example

```
type  
  rec = record  
    ch : char;  
    s : string (10);  
    x : real;  
  end;
```

```

var
  a : array [1..20] of rec;
  b : packed array [1..10] of rec;

begin
  ...
  ...
  pack (a, 10, b);
  ...
  ...
end;

```

An error occurs if the number of elements in the packed array is greater than the number of elements in the source array starting at the specified element to the end of the array.

Round — Rounds to Nearest Integer

The **round** function rounds its argument x and returns an integer value. If the result of rounding the argument x cannot be stored in an integer variable, the maximum negative longint value is returned.

For $x \geq 0$, the result is `trunc(x+0.5)`. For $x < 0$, the result is `trunc(x-0.5)`.

The **round** function is defined as:

```
function round (x);
```

x

is a value parameter of type real or **shortreal**.

Scalar Conversion — Converts Integer to Scalar

Every type identifier for an enumerated scalar or subrange scalar can be used as a function that converts an integer into a value of the enumerated scalar. These functions are the inverse of the **ord** function.

The function is defined as:

```
function type-id (i);
```

i

is an integer valued expression that is converted to an enumerated scalar.

Example

```
type
  days = (sun, mon, tues, weds, thurs, fri, sat);
var
  today : days;
begin
  today := days(0); (* sun *)
  today := days(5); (* fri *)
  ...
end;
```

Str — Converts to String

The **str** function converts either a char or packed array[1..n] of char to a string. RT PC VS Pascal implicitly converts a string to a char or packed array[1..n] of char on assignment, but all other conversions require the conversion to be stated explicitly.

The **str** function is defined as:

```
function str (x);
```

x

is a char or packed array[1..n] of char expression.

Trunc — Truncates to Nearest Integer

The **trunc** function truncates its argument *x* to the nearest integer. The *x* must be of type real or shortreal. If the result of truncating the argument *x* cannot be stored in an integer variable, the maximum negative longint value is returned.

For $x \geq 0$, the result is the largest integer $\leq x$. For $x < 0$, the result is the smallest integer $\geq x$.

The **trunc** function is defined as:

```
function trunc (x);
```

x

is a value parameter of type real or shortreal.

Unpack — Copies Packed Array to Unpacked Array

The **unpack** procedure copies elements from the packed source array to the unpacked target array starting with a specified element of the target array. The type of the elements of the two arrays must be identical.

The **unpack** procedure is defined as:

```

procedure unpack(
  const source : packed__array__type;
  var target : array-type;
    index : index__of__target);

```

source
is a packed array.

target
is an array variable.

index
is an expression which is compatible with the index of *target*.

Example

```

type
  rec = record
    ch : char;
    s  : string (10);
    x  : real;
  end;

var
  a : array [1..20] of rec;
  b : packed array [1..15] of rec;

begin
  ...
  ...
  unpack (b, a, 5);
  ...
  ...
end;

```

An error occurs if the number of elements in the packed array is greater than the number of elements in the unpacked array. The numbering starts with the specified element and continues until the end of the array.

Miscellaneous Low-Level Routines

This section describes various miscellaneous routines. The following list briefly summarizes these routines.

addr returns the memory location of a variable.

sizeof returns the size of a variable in bytes.

Addr — Returns Memory Location of a Variable

The **addr** function returns the location in memory of a specified variable. The specified variables include de-referenced pointers, subscripted variables, and fields of records.

The **addr** function is defined as:

```
function addr(v);  
or  
function addr(v): integer;
```

v
is an identifier that has been declared as a variable.

Sizeof — Determines Size of Data Element or Type

The **sizeof** function returns the number of bytes that a variable or type is allocated. The **sizeof** function is particularly useful in performing block input/output, where the number of bytes to transfer must be known. Note that **sizeof** returns an integer value.

The **sizeof** function is defined as:

```
function sizeof(identifier): integer;  
or  
function sizeof(s : variant record;  
               t1,t2.. : tags): integer;
```

identifier

is a variable name or a type identifier.

s

indicates a record type which has a variant part.

t1,t2..

are tags to variant fields of record *s*.

If the second form of the **sizeof** function is used, **sizeof** gives the size of that variant. If no tag fields are specified, **sizeof** returns the largest variant record.

Control Routines

The control procedures which are described in detail in subsequent sections are listed in the following summary.

clock	returns program execution time.
datetime	returns the current time and date.
halt	terminates a program and returns a termination value.
parms	returns a string associated with the main program.
retcode	sets the return code when leaving the main program.
return	permits exit from a procedure or function.

Clock — Gets Execution Time

The **clock** procedure returns the number of microseconds the program has been running. The **clock** procedure is defined as:

```
function clock : integer;
```

Datetime — Gets Date and Time

The **datetime** procedure returns the current date and time of day as two **alfa** arrays. The **datetime** procedure is defined as:

```
procedure datetime (  
    var date,  
    time: alfa);
```

The format of the result placed in the first and second parameters is respectively:

mm/dd/yy
HH:MM:SS

mm

is the month expressed as a two-digit value.

dd

is the day of the month.

yy

is the last two digits of the year.

HH

is the hour, using a 24 hour clock, of the specified day.

MM

is the minute of the hour.

SS

is the second of the minute.

Halt — Terminates a Program with Return Value

The **halt** procedure terminates the currently executing program. A value is returned to the host operating system to indicate a successful termination or an error termination.

The **halt** procedure is defined as:

```
procedure halt;
```

Parms — Gets Execution Parameters

The **parms** procedure returns a string that is associated with initial invocation of the RT PC VS Pascal main program. The **parms** procedure is defined as:

```
function parms : string;
```

Retcode — Sets Program Return Code

The **retcode** procedure returns the value of the operand to the system when an **exit** is made from the main program. If this routine is called several times, only the last value specified is returned to the system.

The **retcode** procedure is defined as:

```
procedure retcode (  
    retvalue:integer);
```

retvalue

is the return code passed to the caller of the RT PC VS Pascal program. The *retvalue* code must range between 0..255.

Return – Exits from a Procedure or Function

The **return** procedure permits an exit from a procedure or function. This statement is effectively a goto to an imaginary label after the last statement in the routine being executed.

If the check-function compiler directive is enabled, RT PC VS Pascal ensures that a function is assigned a value prior to the return from that function. If a value is not assigned, a run-time error occurs.

The **return** procedure is defined as:

```
procedure return;
```

Example

```
procedure proca
begin
    . . .
    return;
    . . .
{ Return jumps to here }
end;
```

Additional Standard Routines

This section describes addition features which are standard features with RT PC VS Pascal. The following list summarizes these functions.

hbound	returns upper bound of an index to an array.
highest	returns highest value of type scalar.
lbound	returns lower bound of an index to an array.
lowest	returns lowest value of type scalar.
max	returns maximum value of its argument.
min	returns minimum value of its argument.
picture	returns string representation of a real number formatted to a specification.
pred	determines predecessor value of the argument.
succ	determines successor value of the argument.

Hbound — Returns Upper Bound of an Array

The **hbound** function returns the upper bound of an index to an array. The array may be specified as:

- an identifier which is declared as an array type using the type construct
- a variable which is of an array type.

The value returned is of the same type as the type of the index. The second parameter defines the dimension of the array for which the upper bound is returned. If the second parameter is not specified, it is assumed to be "1". The **hbound** function can also be used with **space** types.

The **hbound** function is defined as:

```
function hbound (v,i);  
or  
function hbound (t,i);
```

v

is a variable which is declared as an array type.

t

is a type identifier declared as an array.

i

is a positive integer-valued constant expression and is optional.

Highest — Returns Highest Value of a Scalar Type

The **highest** function returns the highest value that is in the scalar type. The operand may be either a type identifier or a variable. If the operand is a type identifier, the value of the function is the highest that a variable of that type may be assigned. If the operand is a variable, the value of the function is the highest value that the variable may be assigned.

The **highest** function is defined as:

function highest (s);

s

is an identifier that has been declared as a scalar type, or it is a variable which is of a scalar type.

Lbound — Returns Lower Bound of an Array

The **lbound** function returns the lower bound of an index to an array. The array may be specified as:

- an identifier which is declared as an array type using the type construct
- a variable which is of an array type.

The value returned is of the same type as the type of the index. The second parameter defines the dimension of the array for which the lower bound is returned. If the second parameter is not specified, it is assumed to be "1". The **lbound** function can also be used with **space** types.

The **lbound** function is defined as:

```
function lbound (v,i);  
or  
function lbound (t,i);
```

v

is a variable which is declared as an array type.

t

is a type identifier declared as an array.

i

is a positive integer-valued constant expression and is optional.

Lowest — Returns Lowest Value of a Scalar Type

The **lowest** function returns the lowest value that is in the scalar type. The operand may be either a type identifier or a variable. If the operand is a type identifier, the value of the function is the lowest that a variable of that type may be assigned. If the operand is a variable, the value of the function is the lowest value that the variable may be assigned.

The **lowest** function is defined as:

```
function lowest (s);
```

s

is an identifier that has been declared as a scalar type, or it is a variable which is of a scalar type.

Max — Returns Maximum Value of Scalars

The **max** function returns the maximum value of two or more parameters. The parameters may be of any scalar type, including real. They may be a mixture of integer and real expressions. In this case the result is of type real. In all other cases, the parameters must conform to parameter types.

The **max** function is defined as:

```
function max (e0..en);
```

e0..en

is an expression of a scalar type. All parameters must be of the same type except where noted.

Min — Returns Minimum Value of Scalars

The **min** function returns the minimum value of two or more parameters. The parameters may be of any scalar type, including real. They may be a mixture of integer and real expressions, in which case, the result is of type real.

The **min** function is defined as:

```
function min (e0..en);
```

e0..en

is an expression of a scalar type. All parameters must be of the same type except where noted.

Picture — Formats According to Picture Value

The **picture** function returns the string representation of a real number formatted according to a specification. A picture specification may consist of two fields: a decimal field and an exponent field. The decimal field is always required. That exponent field is optional.

The required decimal field may consist of two subfields: a required integer subfield and an optional fractional subfield.

The **picture** function is defined as:

```
function picture(  
  const p : string;  
    r : real) : string(100);  
external;
```

p
is a picture specification.

r
is the number to be formatted.

The **picture** characters may be specified in lowercase. They are grouped into the following categories:

- Digit and decimal point specifier
 - 9** specifies that the associated position in the data item is to contain a decimal digit.
 - v** divides the decimal field into two subfields: integer and fractional. This character specifies that a decimal point is assumed at this position in the associated data item. It does not specify that an actual decimal point is to be inserted. The integer and fractional parts of the assigned value are aligned on the **v** character. An assigned value may be truncated or extended with zero digits

at either end. If a **v** character does not appear, a **v** is assumed at the right end of the decimal field.

- Zero suppression characters

z specifies a conditional digit position in the character string value and may cause a leading zero to be replaced with a blank.

***** specifies a conditional digit position in the character string value and may cause a leading zero to be replaced with an asterisk.

- Insertion character

Insertion characters are added into corresponding positions in the output string provided that zero suppression is not occurring. If zeros are being suppressed when an insertion character is encountered, a blank or an asterisk is inserted in the corresponding place in the output string, depending on whether the zero suppression character is **z** or *****.

, causes a comma to be inserted into the associated position of the output string.

. causes a point (.) to be inserted into the output string. The character never causes point alignment in the number. The function is served by the character **v**.

b causes a blank to be inserted into the associated position of the output string.

- Signs and currency symbols

The sign and currency characters (**+**, **-**, **s**, **\$**) may be used in either a static or a drifting manner. The static use specifies that a sign, a currency symbol, or a blank always appears in the associated position. The drifting use specifies that leading zeros are to be suppressed. A drifting character is specified by multiple use of that character in a picture field.

+ specifies a plus sign character (**+**) if the number is greater than or equal to zero. Otherwise, it specifies a blank.

- specifies a minus sign character (-) if the number is less than zero. Otherwise, it specifies a blank.
- s specifies a plus sign character (+) if the number is greater than or equal to zero. Otherwise, it specifies a minus sign character (-).
- \$ specifies a dollar sign character (\$).
- Exponent specifiers

The following characters are used to delimit the exponent field of a picture specification. The exponent field must always be the last field.

- e specifies that the associated position contains the letter e which indicates the start of the exponent field.
- k specifies that the exponent field appears to the right of the associated position. It does not specify a character data item.

Examples

Picture	Value	Result
'999v.99'	123.456	'123.46'
's9v.9999es99'	1.23456	'+1.2346e+00'
'zzzv.99'	0.12	' .12'

Pred — Determines Predecessor of Value

The **pred** function accepts an argument that is any scalar type except real or shortreal. The result of **pred** is the predecessor value of the argument, if such a predecessor value exists. If *x* does not have a successor value, **pred** is undefined.

The **pred** function is defined as:

```
function pred(x);
```

x

is an any scalar type except real or shortreal.

Succ — Determines Successor of Value

The **succ** function accepts an argument that is any scalar type except real or shortreal. The result of **succ** is the successor value of the argument, if such a successor value exists. If x does not have a successor value, **succ** is undefined.

The **succ** function is defined as:

```
function succ(x);
```

x

is an any scalar type except real or shortreal.

Appendix A. Built-In Procedure and Function Summary

This section briefly lists and describes all the "built-in" procedures and functions available in RT PC VS Pascal.

Option	Description	Procedure/ Function	IBM Mode	ANSI Mode
abs	computes absolute value of its argument.	F	•	•
addr	returns the memory location of available.	F	•	
arctan	computes arctangent of the argument.	F	•	•
chr	converts an integer to a character.	F	•	•
clock	returns program execution time.	P	•	
close	removes the association of a file variable with an external file.	P	•	
cols	determines the current column number on an output file.	F	•	

Figure A-1 (Part 1 of 8). RT PC VS Pascal Procedures and Functions

Option	Description	Procedure/ Function	IBM Mode	ANSI Mode
compress	replaces multiple blanks with one blank.	F	•	
cos	computes cosine of the argument.	F	•	•
datetime	returns the current time and date.	P	•	
delete	deletes characters from a string.	F/P	•	
dispose	is responsible for freeing or releasing storage back to the common storage pool.	P	•	•
eof	returns a value of true if a read encounters an end-of-file	F	•	•
eoln	returns a value of true if the text file position is at an end-of-line character.	F	•	•
exp	computes exponential of the argument.	F	•	•
float	converts an integer to a real.	F	•	

Figure A-1 (Part 2 of 8). RT PC VS Pascal Procedures and Functions

Option	Description	Procedure/ Function	IBM Mode	ANSI Mode
get	obtains the next element from a file.	P	•	•
halt	terminates a program and returns a termination value.	P	•	
hbound	returns upper bound of an index to an array.	F	•	
highest	returns highest value of type scalar.	F	•	
index	scans for a pattern within a string and returns the starting index of the first instance of the pattern in the source string.	F	•	
itohs	converts an integer into a hexadecimal string.	F	•	
lbound	returns lower bound of an index to an array.	F	•	
length	determines the current dynamic length of a string.	F	•	
ln	computes natural logarithm of the argument.	F	•	•

Figure A-1 (Part 3 of 8). RT PC VS Pascal Procedures and Functions

Option	Description	Procedure/ Function	IBM Mode	ANSI Mode
lowest	returns lowest value of type scalar.	F	•	
lpad	pads or truncates the string variable on the left.	P	•	
ltrim	removes the leading from a string.	F	•	
mark	provides a means to "remember" the current top of the heap.	P	•	
max	returns maximum value of its argument.	F	•	
maxlength	returns the declared static length of a string.	F	•	
min	returns minimum value of its argument.	F	•	
new	is responsible for allo- cating storage.	P	•	•
odd	tests for odd or even	F	•	•
ord	converts a value of an ordinal type to an integer.	F	•	•

Figure A-1 (Part 4 of 8). RT PC VS Pascal Procedures and Functions

Option	Description	Procedure/ Function	IBM Mode	ANSI Mode
pack	copies elements from an unpacked array to a packed array.	P	•	•
page	skips to the top of a new page on a text or interactive file.	P	•	
parms	returns a string associated with the main program.	P	•	
picture	returns string representation of a real number formatted to a specification.	F	•	
pred	determines predecessor value of the argument.	F	•	•
put	appends the value of the buffer variable.	P	•	•
random	computes a random number.	F	•	

Figure A-1 (Part 5 of 8). RT PC VS Pascal Procedures and Functions

Option	Description	Procedure/ Function	IBM Mode	ANSI Mode
read and readln	read character strings representing numbers from a text file and convert them into their internal representations.	P	•	•
readstr	converts a string to a different type.	P	•	
release	releases memory from a previously marked point.	P	•	
reset	opens the file and positions the file pointer at the beginning of the file.	P	•	•
retcode	sets the return code when leaving the main program.	P	•	
return	permits exit from a procedure or function.	P	•	
rewrite	creates a new file of a specified name and discards an existing file of the same name.	P	•	•

Figure A-1 (Part 6 of 8). RT PC VS Pascal Procedures and Functions

Option	Description	Procedure/ Function	IBM Mode	ANSI Mode
round	rounds its argument so that the result is of type integer.	F	•	•
rpadd	pads or truncates the string variable on the right.	P	•	
scalar conversion	converts an integer to an enumerated scalar.	F	•	
seek	supports random access to files of specific types.	P	•	
sin	computes sin of the argument.	F	•	•
sizeof	returns the size of a variable in bytes.	F	•	
sqr	computes square of the argument.	F	•	•
sqrt	computes square root of the argument.	F	•	•
str	converts a char to a string.	F	•	
substr	returns specified substring from a string.	F	•	

Figure A-1 (Part 7 of 8). RT PC VS Pascal Procedures and Functions

Option	Description	Procedure/ Function	IBM Mode	ANSI Mode
succ	determines successor value of the argument.	F	•	•
termin	opens a designated file for interactive input from the terminal.	P	•	
termout	opens a designated file for output from the terminal.	P	•	
token	scans a string and returns it as an alpha.	P	•	
trim	removes the trailing blanks from a string.	F	•	
trunc	truncates its argument to the nearest integer.	F	•	•
unpack	copies elements from a packed array to an unpacked array.	P	•	•
update	opens a designated file for both input and output updating.	P	•	
write and writeln	append character strings to a text file.	P	•	•
writestr	converts an expression into character data.	P	•	

Figure A-1 (Part 8 of 8). RT PC VS Pascal Procedures and Functions

Index

A

- abs function 8-23
- absolute value 8-23
- accessing variables 3-4
- addition
 - operators 4-6
- addr function 8-36
- alfa type 2-7
- alpha type 2-7
 - simple 2-7
- ANSI mode 1-2
- arctan function 8-24
- arctangent 8-24
- arguments 7-20
- arithmetic functions 8-25
- arithmetic routines 8-23-8-27
 - abs 8-23
 - arctan 8-24
 - cos 8-24
 - exp 8-24
 - ln 8-25
 - random 8-25
 - sin 8-26
 - sqr 8-26
 - sqrt 8-27
- array
 - hbound 8-42
 - lbound 8-43
 - pack an array 8-31
 - unpacking a packed array 8-34
- array constants 2-27
- array types 1-5, 2-8
 - defined 1-5

- index-types 2-8
- ASCII character set 1-12
- assert statements 5-14
- assignment compatibility 2-23
 - types 2-24
- assignment statements 1-9, 5-2
- attributes
 - See calling attributes

B

- back reference tag field 2-16
- basic types, listed 1-5
- begin-end statements 5-5
- binary constants 2-26
- binary operators 4-11
- blanks (spaces) 1-19
- block 7-6
 - scope of 7-6
- boolean attribute routines 8-27
 - odd 8-27
- boolean expressions 4-11
- boolean type 2-3
 - writing 6-21
- booleans, compared 4-9
- buffer variable 6-1
- built-in functions
 - defined 8-1
 - listed A-1, A-8
- built-in procedures
 - defined 8-1
 - listed A-1, A-8

C

- calling attributes 7-26
 - external 7-27
 - FORTRAN 7-27
 - forward 7-27
 - main 7-26
 - reentrant 7-27
- case statements 5-5
- char data
 - reading 6-13
 - reading packed array 6-14
 - writing 6-21
- char type 2-3
- character data, reading 6-13, 6-14
 - packed array of char data 6-14
- character deletion from strings 8-4
- character set 1-12
- chr function 8-28
- clock procedure 8-38
- close a file 6-3
- close procedure 6-3
- cols function 6-8
- comments 1-19
- compilation unit 7-1
 - definition of 7-3
 - examples of 7-2-7-3
 - program 7-3
 - segment 7-3
 - segment module 7-4
 - terms 7-3
- compile-time constant expressions 4-11
- compiler directives 7-1
- component variable 3-5
- compress function 8-3
- concepts 1-4-1-20
 - declarations 1-4
 - statements 1-9

- conformant string parameters 7-23
- const 7-23
 - pass by 7-23
- constant expression 2-25
- constant identifier 2-24
- constants
 - binary 2-26
 - defined by 2-24
 - definition 7-15
 - hexadecimal 2-26
 - literal 2-24
 - predefined 2-25
 - string 1-16
 - structured 2-27
 - unsigned 4-1
- continue statement 5-14
- control routines 8-37-8-41
 - clock 8-38
 - datetime 8-38
 - halt 8-39
 - parms 8-39
 - retcode 8-40
 - return 8-40
- control statements 5-14-5-17
 - assert 5-14
 - continue 5-14
 - goto 5-15
 - leave 5-16
- cos function 8-24
- cosine, computing 8-24
- creating a file 6-6

D

- data types
 - assignment-compatible 2-24
 - constants 2-24
 - defined 2-1
 - identical 2-23

- non-comparable 4-10
- pointer 2-21
- simple 2-4
- standard 2-2
- structured 2-8
- syntax 2-3
- datetime procedure 8-38
- dead code elimination 4-12
- declarations 1-4, 7-6, 7-8
 - constant definitions 7-15
 - def/ref declarations 7-8
 - definition of 7-6
 - function 7-16, 7-19
 - label 7-8
 - procedure 7-16, 7-17
 - space declarations 7-12
 - static declarations 7-10
 - type definitions 7-16
 - value declarations 7-11
 - variable 7-16
- def declarations 7-8
- delete function 8-4
- discriminant 2-13
- dispose procedure 8-17
- dummy parameters 7-21
- dynamic variables 1-9, 3-3, 3-4

E

- element 4-1
- empty statements 5-7
- end-of-file 6-3
- end-of-line 6-9
- entire variable 3-5
- enumerated type 2-4
- eof function 6-3
- eoln function 6-9
- even or odd testing 8-27
- exp function 8-24
- explicitly declared variables 1-8, 2-21, 3-2

- exponential value 8-24
- expressions
 - compile-time constant 4-11
 - defined 4-1
 - element 4-1
 - expression 4-2
 - factor 4-1
 - operators in 4-2-4-8
 - order of evaluation 4-2
 - set constructor 4-1
 - simple-expr 4-2
 - term 4-2
 - unsigned constant 4-1
- external attribute 7-18, 7-20, 7-27

F

- factor 4-1
- false constant 2-25
- fields, offset qualifications 2-17
- file buffer variable 6-1
- file buffers, referencing 3-8
- file handling routines 6-2-6-7
 - close 6-3
 - eof 6-3
 - get 6-4
 - put 6-5
 - reset 6-5
 - rewrite 6-6
 - update 6-6
- file type 2-19
 - defined 1-8
 - random 1-8
 - sequential 1-8
- fixed records 1-5
- float function 8-29
- for-do statements 5-8
- formal parameters 1-10, 3-2, 3-3, 7-21
 - conformant string 7-22

- function 7-21
- pass by const 7-22
- procedure 7-21
- reference 7-21
- value 7-21
- variable 7-21
- FORTRAN 7-20
- FORTRAN attribute 7-18, 7-27
- forward attribute 7-18, 7-20, 7-27
- function declarations 7-16
 - calling attributes 7-26
 - definition of 7-19
 - parameter examples 7-24
 - parameters 7-20
- function parameter 1-11
- function parameter examples 7-24
- functions 6-1
 - abs 8-23
 - addr 8-36
 - arctan 8-24
 - arithmetic 8-25
 - assignments to 5-2
 - built-in, defined 8-1
 - chr 8-28
 - cols 6-8
 - cos 8-24
 - defined 1-11
 - delete 8-4
 - eof 6-3
 - eoln 6-9
 - exp 8-24
 - external 7-18, 7-20
 - float 8-29
 - forward 7-18, 7-20
 - hbound 8-42
 - highest 8-43
 - itohs 8-29
 - lbound 8-43
 - length 8-6
 - listing A-1, A-8
 - ln 8-25
 - lowest 8-44

- max 8-45
- maxlength 8-8
- min 8-45
- odd 8-27
- ord 8-30
- picture 8-46
- pred 8-48
- random 8-25
- recursive 1-12
- round 8-32
- scalar conversion 8-33
- sin 8-26
- sizeof 8-36
- sqr 8-26
- sqrt 8-27
- str 8-33
- substr 8-11
- succ 8-49
- text file handling 6-8-6-22
- trunc 8-34
- functions, listed A-1, A-8

G

- get component from a file 6-4
- get procedure 6-4
- global variables 3-3
- goto statements 5-15

H

- halt procedure 8-39
- hbound function 8-42
- heap 8-16
 - marking position of 8-17
- hexadecimal
 - constants 2-26

highest function 8-43
host 7-3
 definition of 7-3

I

IBM mode 1-1
identical types 2-23
identifiers 1-13, 7-6
 block scope 7-6
 segment scope 7-6
identity (+) sign 4-4
if-then-else statements 5-9
index function 8-5
indexed variables, referencing 3-5
input and output 6-1
input file 7-7
input variable 3-2
integer data, reading 6-12
integer element 6-20
integer type 2-2
interactive type 2-20
 structured 2-20
itohs function 8-29

L

label declarations 7-8
labels 1-16
language constructs, listed 1-12
lbound function 8-43
leave statements 5-16
length function 8-6
lexical constructs, listed 1-12
lifetime of variables 3-3-3-4
literal constant 2-24
ln function 8-25
local variables 3-3

logarithm 8-25
 natural 8-25
low-level routines 8-36-8-37
 addr 8-36
 sizeof 8-36
lowest function 8-44
lpad procedure 8-7
ltrim function 8-7

M

main attribute 7-18, 7-20, 7-26
mark procedure 8-17
max function 8-45
maxint constant 2-26
maxlength function 8-8
maxreal constant 2-26
memory, releasing 8-21
methods of presentation 1-3
min function 8-45
minint constant 2-26
minreal constant 2-26
modes
 ANSI 1-2
 IBM 1-1
multiplication
 operators 4-4

N

negation (-) sign 4-4
new procedure 8-18
nil pointer 1-9, 2-21
not operator 4-2, 4-3
numbers 1-14
 unsigned integer 1-14
 unsigned number 1-15

unsigned real 1-15

O

.o files 7-1
odd function 8-27
odd or even testing 8-27
offset qualifications of fields 2-17
open an existing file 6-5
opening a file 6-6
operators in expressions
 addition 4-6
 multiplication 4-4
 not 4-2, 4-3
 order of evaluation 4-2
 relational 4-7
 sign 4-4
 unary 4-4
ord function 8-30
ordinal number 8-30
ordinal type 2-4
out-of-range values 4-11
output and input 6-1
output file 7-7
output variable 3-2

P

pack procedure 8-31
packed array of char, writing 6-22
packing structured types 2-8
page procedure 6-10
parameters 1-10, 7-20
 conformant string 7-23
 dummy 7-21
 examples 7-24
 formal 7-21

formal parameters 3-2
function 7-24
pass by const 7-23
procedure 1-10, 7-24
value 1-10, 7-22
variable 1-10, 7-22
parametric procedures and functions 7-24
parms procedure 8-39
pass by const parameter 7-22
picture function 8-46
 decimal point specifier 8-46
 digit specifier 8-46
 exponent specifiers 8-48
 insertion characters 8-47
 signs and currency symbols 8-47
 zero suppression characters 8-47
pointer types 2-21
pointers 1-9
 nil 1-9
pointers, comparison 4-9
pred function 8-48
predeclared variables 3-2
 input 3-2
 output 3-2
 stderr 3-2
predefined constants 2-25
 false 2-25
 maxint 2-26
 maxreal 2-26
 minint 2-26
 minreal 2-26
 true 2-25
presentation methods 1-3
procedure call statements
 See procedure reference statements
procedure declarations 1-10, 7-16
 calling attributes 7-26
 definition of 7-17
 parameter examples 7-24
 parameters 7-20
procedure parameter 1-11
procedure parameter examples 7-24

- procedure reference statements 5-3
- procedure statement 1-9
- procedures
 - actual parameter 1-10
 - built-in, defined 8-1
 - clock 8-38
 - close 6-3
 - compress 8-3
 - datetime 8-38
 - dispose 8-17
 - formal parameters 1-10, 3-2
 - FORTRAN 7-27
 - general file handling 6-7
 - get 6-4
 - halt 8-39
 - index 8-5
 - listing A-1
 - lpad 8-7
 - ltrim 8-7
 - main 7-26
 - mark 8-17
 - new 8-18
 - pack 8-31
 - page 6-10
 - parms 8-39
 - procedure parameter 1-10
 - put 6-5
 - read 6-10
 - readln 6-10
 - readstr 8-9
 - recursive 1-12
 - reentrant 7-27
 - release 8-21
 - reset 6-5
 - retcode 8-40
 - return 8-40
 - rewrite 6-6
 - rpadd 8-11
 - seek 6-16
 - statements 1-9
 - termin 6-16
 - termout 6-17

- text file handling 6-8-6-10, 6-22
- token 8-13
- trim 8-14
- unpack 8-34
- update 6-6
- value parameter 1-10
- variable parameter 1-10
- write 6-17
- writeln 6-17
- writestr 8-15
- procedures, listed A-1, A-8
- program heading 7-4
- program structure
 - compilation unit 7-1
 - program heading 7-4
- program, defined 7-3
- put procedure 6-5

R

- random access to typed files 6-15
- random function 8-25
- read procedures 6-10
 - char data 6-13
 - from a file of any type 6-13
 - integer data 6-12
 - packed array of char 6-14
 - real data 6-13
 - shortreal data 6-13
 - string data 6-13
 - variables with a length 6-14
- readln procedure 6-10
- readstr 8-9
- real data, reading 6-13
- real data, writing 6-20
- real type 2-2
 - notes 2-2
- record constants 2-27
- record fields, referencing 3-7

- record types 1-5, 2-11
 - defined 1-5
 - field offset qualifications 2-17
 - fixed records 1-5
 - packed 2-17
 - tag field 1-6
 - variant records 1-5
- recursive functions 1-12
- recursive procedures 1-12
- reentrant attribute 7-18, 7-20, 7-27
- ref declarations 7-8
- referenced variable 3-4
- relational operators 4-7
 - comparison
 - of booleans 4-9
 - of scalars 4-8
 - set 4-10
 - string 4-9
- release procedure 8-21
- repeat-until statements 5-11
- repetition expression 2-27
- replaces component to a file 6-5
- reserved words 1-16
- reset procedure 6-5
- retcode procedure 8-40
- return procedure 8-40
- return value
 - set program return code 8-40
 - terminating a program with 8-39
- rewrite procedure 6-6
- round function
 - to nearest integer 8-32
- rpadd procedure 8-11

S

- scalar conversion function 8-33
- scalar conversions 2-5
- scalar subrange element 6-20
- scalar types 1-5, 2-4
 - comparison 4-8
 - defined 1-5
 - scalar conversions 2-5
- scalar, maximum value 8-45
- scalar, minimum value 8-45
- scope of identifier 1-10
- seek procedure 6-16
- segment 7-3, 7-4
 - definition of 7-3
 - scope of 7-6
- set constructor 4-1
- set types 1-8, 2-18
 - comparison 4-10
 - defined 1-8
- shortreal data, writing 6-20
- shortreal data, reading 6-13
- shortreal type 2-2
 - notes 2-2
- sign operators
 - identity (+) 4-4
 - negation (-) 4-4
- simple types 2-4-2-7
 - alfa type 2-7
 - alpha type 2-7
 - scalar 2-4
 - subrange types 2-6
- simple-expr 4-2
- sin function 8-26
- sine 8-26
- sizeof function 8-36
- skip to new page 6-10
- space declarations 7-12
 - referencing 7-14
- spaces (blanks) 1-19

- sqr function 8-26
- sqrt function 8-27
- square of a number 8-26
- square root 8-27
- standard error output file (stderr) 7-7
- standard files
 - input 7-7
 - output 7-7
 - stderr 7-7
- standard routines 8-41-8-49
 - hbound 8-42
 - highest 8-43
 - lbound 8-43
 - lowest 8-44
 - max 8-45
 - min 8-45
 - picture 8-46
 - pred 8-48
 - succ 8-49
- standard types 2-2
 - boolean 2-3
 - char 2-3
 - integer 2-2
 - real 2-2
 - shortreal 2-2
 - text 2-3
- statement labels 5-1
 - scope of 5-1
- statements 1-9
 - assignment 5-2
 - labels 5-1
 - procedure reference 5-3
- static declarations 7-10
- static variables 3-4
- stderr variable 3-2
- storage allocation 8-18
- storage allocation procedures 8-16-8-22
 - dispose 8-17
 - mark 8-17
 - new 8-18
 - release 8-21
- storage, freeing 8-17

- str function 8-33
- string data, reading 6-13
- string data, writing 6-22
- string manipulation routines 8-1-8-16
 - compress 8-3
 - delete 8-4
 - index 8-5
 - length 8-6
 - lpad 8-7
 - ltrim 8-7
 - maxlength 8-8
 - readstr 8-9
 - rpadd 8-11
 - substr 8-11
 - token 8-13
 - trim 8-14
 - writestr 8-15
- string types 1-8, 2-10
 - defined 1-8
- stringptr 2-22
- strings
 - comparison 4-9
 - constants 1-16
 - converting to 8-33
 - converts type 8-9
 - formats according to picture value 8-46
 - length 8-6, 8-8
 - pad or truncate on the left 8-7
 - pad or truncate on the right 8-11
 - referencing 3-6
- structured constants 2-27
 - array constants 2-27
 - record constants 2-27
- structured statements 1-10, 5-4-5-13
 - begin-end 5-5
 - case 5-5
 - empty 5-7
 - for-do 5-8
 - if-then-else 5-9
 - repeat-until 5-11

- while-do 5-11
- with 5-12
- structured types 1-5, 2-8-2-20
 - array types 2-8
 - defined 1-5
 - record types 2-11
 - set types 2-18
 - string types 2-10
 - structuring methods 2-8
- subrange types 2-6
- substr function 8-11
- succ function 8-49
- symbols 1-16
 - reserved words 1-16
 - special symbols 1-17
- syntax conventions 1-3

T

- tag field 1-6, 2-27
- term 4-2
- termin procedure 6-16
- termout procedure 6-17
- terms 1-4-1-20
 - declarations 1-4
 - statements 1-9
- text file handling routines 6-8-6-22
 - cols 6-8
 - eoln 6-9
 - page 6-10
 - read 6-10
 - readln 6-10
 - seek 6-16
 - termin 6-16
 - termout 6-17
 - write 6-17
 - writeln 6-17
- text type 2-3
 - standard 2-3
- token 8-13

- trim 8-14
- true constant 2-25
- trunc function 8-34
- truncating
 - to nearest integer 8-34
- type definitions 7-16
- type identity 2-23
- types
 - See data types

U

- unary operators 4-4, 4-11
 - identity (+) 4-4
 - negation (-) 4-4
- unit
 - programs 7-2-7-3
- unpack procedure 8-34
- unsigned integer 1-14
- unsigned number 1-15
- unsigned real 1-15
- update procedure 6-6

V

- value conversion routines 8-28, 8-35
 - chr 8-28
 - float 8-29
 - itohs 8-29
 - ord 8-30
 - pack 8-31
 - round 8-32
 - scalar conversion 8-33
 - str 8-33
 - trunc 8-34
 - unpack 8-34
- value declarations 7-11

- value parameters 1-10, 7-22
- variable declarations 7-16
- variable parameters 1-10, 7-22
- variables 3-1
 - assignments to 5-2
 - declaring 3-1
 - dummy 7-21
 - dynamic 1-9, 3-3, 3-4
 - establishing 3-2
 - explicitly declared 1-8, 3-2
 - file buffer 6-1
 - global 3-3
 - indexed 3-5
 - lifetime of 3-3-3-4
 - local 3-3
 - pointer variables 3-8
 - predeclared 3-2
 - referencing 3-4-3-9
 - static 3-4
 - strings 3-6
- variables, accessing 3-4
- variables, reading 6-14
- variables, referencing
 - component variables 3-5
 - entire variables 3-5
 - fields of records 3-7
 - file buffers 3-8

- indexed variables 3-5
- pointer referenced variables 3-8
- strings 3-6
- variant records 1-5, 2-13

W

- while-do statements 5-11
- with statements 5-12
- write parameters 6-18
- write procedure 6-17
 - boolean data 6-21
 - char data 6-21
 - files of any type 6-22
 - integer element 6-20
 - packed array of char 6-22
 - parameters 6-18
 - real data 6-20
 - scalar subrange element 6-20
 - shortreal data 6-20
 - string data 6-22
- writeln procedure 6-17
 - parameters 6-18
- writestr 8-15
- writing to a file of any type 6-22



The IBM RT PC

Reader's Comment Form

IBM RT PC VS Pascal
Reference Manual

SH23-0128-0

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

For prompt resolution to questions regarding set up, operation, program support, and new program literature, contact the authorized IBM RT PC dealer in your area.

Comments:

Tape

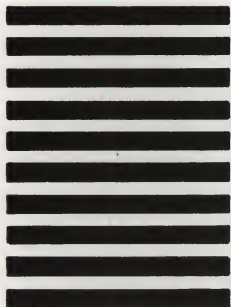
Please Do Not Staple

Tape

Cut or Fold Along Line

Fold and tape

Fold and tape



International Business Machines Corporation
Department 79L, Building 4
Commerce Park & Eagle Road
Danbury, Connecticut 06810

POSTAGE WILL BE PAID BY ADDRESSEE:

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES





© IBM Corp. 1987
All rights reserved.

International Business Machines Corporation
Department 79L, Building 4
Commerce Park and Eagle Road
Danbury, CT 06810

Printed in the
United States of America

SH23-0128



SH23-0128-00

